



Wouter van Toll
Arjan Egges
Jeroen D. Fokker

Learning C#

by Programming Games

Second Edition



Springer

Learning C# by Programming Games

Wouter van Toll • Arjan Egges • Jeroen D. Fokker

Learning C# by Programming Games

Second Edition

 Springer

Wouter van Toll
Rennes, France

Arjan Egges
Utrecht, The Netherlands

Jeroen D. Fokker
Department of Information and
Computing Sciences
Utrecht University
Utrecht, The Netherlands

ISBN 978-3-662-59251-9 ISBN 978-3-662-59252-6 (eBook)
<https://doi.org/10.1007/978-3-662-59252-6>

© Springer-Verlag GmbH Germany, part of Springer Nature 2013, 2019

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Cover design: Cover design based on a game sprite designed by Heiny Reimes. Reused by permission.

This Springer imprint is published by the registered company Springer-Verlag GmbH, DE, part of Springer Nature. The registered company address is: Heidelberger Platz 3, 14197 Berlin, Germany

Preface

Introduction

If you are reading this, then we assume that you are interested in learning how to develop your own computer games. If you do not yet know how to program, don't worry! Developing computer games is actually a perfect way to learn how to program in modern programming languages. This book will teach you how to program in C# without requiring any previous programming experience. It does so through the creation of computer games.

In our opinion, C# is the language of choice if you want to learn how to program. The language is very well structured, it contains many modern features, and it is used a lot for game development and software development in general. Compared to Java (another popular programming language), some things are slightly easier to understand in C#. Also, once you understand how C# works, it is relatively easy to move to C++, which is still one of the most important programming languages in the game industry.

This book gives you a thorough introduction to C# and object-oriented programming in general. The unique aspect of this book is that it uses the structure and elements of computer games as a basis. For instance, the book contains chapters on dealing with player input, game objects, game worlds, levels, animation, physics, and intelligence. Along the way, you will learn about many programming concepts, such as variables, objects, classes, loops, collections, and exception handling.

We'll also discuss various aspects of software design that are useful for game development. For example, we propose useful techniques for managing levels and game states, as well as a hierarchy of game objects that together form an interactive game world. Furthermore, we will show a number of commonly used techniques in games, such as drawing layers of sprites, rotating, scaling and animating sprites, dealing with physics, handling interaction between game objects, and creating nice visual effects.

Throughout the book, you will create four games that gradually become more complicated. We have chosen different types of games to show the various aspects of game development. We start with a simple shooting game, we move on to puzzle games consisting of multiple levels, and we conclude with a full-fledged platform game with animation, game physics, and intelligent enemies.

For each of these games, the book invites you to write the code yourself. After all, the best way to learn how to program is to simply *do it*. Along with this book, we've created many example projects that you can use for reference.

Target Audience

This book is aimed at people who:

- are interested in game programming,
- have little to no programming experience so far,
- want to know how programming *really* works in general, so that they can use this knowledge in other places as well.

Using games as a guideline, this book will teach you the most important elements of object-oriented programming in C#. You'll also learn about programming on a more abstract level, so that you can easily teach yourself other programming languages in the future.

Due to this setup, the book is also very useful as a textbook for an introductory programming course on university level (see the "For Teachers" section). However, our target audience is not limited to students: if you recognize yourself in the bullet-point list above, then you can benefit from this book, no matter what your background is.

Structure of This Book

This book is divided into five parts. Each part consists of multiple chapters, and each chapter has its own collection of example programs in which you learn new programming concepts. You can download these example programs from the book's website.

Overall, the five parts of this book have the following contents:

- Part **I** serves as an introduction to programming in general, and it shows how games are generally developed. It introduces basic programming concepts such as variables, methods, and parameters. It also discusses the game loop: the basis of every game program.
- In Part **II** of the book, you will develop your first game called Painter. You will first learn how to deal with game assets such as sprites and sounds. Then, you will learn about the **if**-instruction and loops such as **for** or **while**. Furthermore, you'll learn about the basics of object-oriented programming. You will also see how to create a game world that consists of multiple interacting game objects. In terms of general programming techniques, this is the most important part of the book. After this part, the games will become more complicated, but they will reuse many of the techniques you've already learned in Part **II**.
- In Part **III**, you will continue setting up this structure of communicating game objects by using *arrays* and *collections*. These programming concepts are very important for the second game, Jewel Jam, which is a pattern recognition game. You will also see how to scale a game to different screen sizes and how to model the game world as a *hierarchy* of objects that can contain each other.
- Part **IV** introduces the puzzle game Penguin Pairs. This game shows you how to deal with different game states such as menus or level selection screens. You'll learn how to read levels from files and how to store the player's progress in a file. You'll also learn how to deal with sprite sheets: larger image files that store multiple "sub-images" for your game. Finally, this part deals with organizing classes into different libraries that you can use across different game projects.
- Finally, Part **V** revolves around the platform game Tick Tick, which reuses the game engine you've been building so far. In this part, you will deal with loading and playing animations, and you will add basic physics to the game world, such as jumping, falling, and handling collisions with other game objects. You'll also learn how to deal with unexpected errors (*exceptions*).

Each chapter (except Chap. 1) ends with a collection of *exercises*. Sometimes, these are general questions that help you get more familiar with the concepts introduced in that chapter. Other exercises invite you to extend the example programs from that chapter, using the things you’ve learned so far. The more advanced exercises are marked with an asterisk (*). Furthermore, in the last chapter of each part (starting in Part II), there’s an exercise that invites you to extend the main game from that part. These are open and creative challenges that you can take as far as you want.

Finally, throughout the book, you will find text in gray boxes. These boxes usually contain “side notes” that go a bit deeper than the main text. In this second edition of the book, we’ve also added *Quick Reference* boxes that each give a short summary of a programming concept.

Required Materials and Tools

To develop the games of this book, you will need to install a few tools on your computer:

- *Visual Studio Community 2017*, a free program by Microsoft for writing and compiling code in C#.
- *MonoGame 3.6*, an open-source library that adds game-specific functionality to the C# language.

On the website of this book, you can find detailed instructions on how to obtain and install these tools. Chapter 1 will also help you further with these preparations.

Note that Visual Studio and MonoGame are updated frequently. By the time you read this book, newer versions of these tools may already be available. Of course, you are free to use these newer versions as well. However, if you want to be sure that you can run our example projects, we advise you to install the versions mentioned above.

If these versions of Visual Studio or MonoGame become so outdated that the book becomes too difficult to use, we will try to update the book’s website to help you out as much as possible.

Along with this book, we supply various materials. All the example programs used in this book are available as Visual Studio projects that you can open, edit, compile, and run yourself. Furthermore, we supply a set of game assets (sprites and sounds) that are used in our example programs. All these additional materials can be found on the accompanying website of this book.

Website — Along with the book itself, there is an accompanying website from which you can download all additional materials. The URL of this website is <http://www.csharpprogramminggames.com/>.

For Teachers: Using This Book for a Programming Course

This book is very suitable as a basis for a game-oriented programming course. Each part of this book is concluded by exercises and challenges. Solutions to the exercises are available on the book’s website. The challenges are generally more complex programming exercises, and they can serve as practical assignments for students following the course. On the accompanying website, a number of additional challenges are available that can be used as a basis for practical assignments as well.

By following the structure of the book throughout the course, students will be introduced to the main aspects of programming in an object-oriented language. Supplementary materials for organiz-

ing such a course are available on the website of this book, www.csharpprogramminggames.com. A sample schedule of a course with 15 sessions and 3 practical assignments is given below:

Session	C# topics	Game topics	Chapters	Deadlines
1	Introduction		1, 2	
2	Types, variables	Game loop	3, 4	
3	if, else , booleans	Sprites, sound, player input	5, 6	
4	Classes, methods, objects	Object interaction	7, 8	
5	for, while , randomness		8, 9	Ass. 1
6	Inheritance	Drawing text	10, 11	
7	Arrays, strings, collections	Screen sizes, grids	12, 13	
8	Recursion	Game worlds	14, 15	
9	Abstract classes, interfaces		16, 17	
10		Game states, sprite sheets	17, 18	Ass. 2
11	File I/O, switch	Tiles, levels	19, 20	
12	Libraries, exceptions		21, 22	
13		Animation, game physics	23, 24	
14		Enemies	25, 26	
15	General questions		All	Ass. 3

New in the Second Edition

In this second edition of the book, we've made a number of key changes, based on our experience with using the book in a university-level game programming course. The text has been rewritten entirely, and chapters have been restructured and reordered. All in all, this second edition should provide an even smoother and more general introduction to (game) programming.

The most important changes in this edition are the following:

- We have updated all example programs to work with Visual Studio Community 2017 and MonoGame 3.6 (instead of the now obsolete XNA Game Studio). Readers can once again follow the material by using the most modern programming tools (at the time of writing this book).
- Throughout this book, the text now invites you to write your *own* code, instead of explaining how the book's example projects work. The example projects are still there as reference points, but you're encouraged to get to those reference points by writing the code yourself. We've added convenient checkpoint markers that indicate when your code should match a certain example project.
- We have made the most important terms easier to find in the text. All terms that appear in the index (at the end of the book) are printed in boldface when they're first explained.
- We have made a clearer distinction between general C# programming concepts and concepts that are specific to game development. For example, primitive types (such as **int** and **string**) and the concept of classes are part of the C# language, but the **Color** class and the game loop are specific to the MonoGame library. If you want to continue with software development in general, it's good to be aware of this distinction.
- We've made programming guidelines easier to recognize. The new "Quick Reference" boxes summarize a programming concept in a way that clearly stands out from the rest of the text. As a result,

compared to the first edition, it is now easier to see the difference between an example and the general lesson that you can learn from it.

- The exercises are now grouped per chapter (instead of per part), and they can now be found directly after their chapter (instead of in an appendix). This way, readers can check more easily whether they understand the material from a particular chapter. The exercises themselves have also been given an update: we've added and changed several exercises to give readers a more general education in C# programming.

If you're already using the first edition of the book for personal reference, then you can keep using it without too many difficulties: both versions of the book still cover the same material overall. In all other cases, we strongly recommend using the second edition: it's a more convenient introduction to (game) programming and a more complete basis for a course. If you are a student and your programming course has been updated to the second edition of the book, then upgrading is especially useful, also because the book's example projects and exercises have changed.

We hope you'll enjoy using this book for gaining your first experience with object-oriented (game) programming. When you have reached the end of this book, you will have a solid basis for becoming a good programmer. What comes after that is up to you, but we hope you'll be inspired to create wonderful games and other programs in the future.

Good luck!

Rennes, France
Utrecht, The Netherlands
Utrecht, The Netherlands

Wouter van Toll
Arjan Egges
Jeroen D. Fokker

Acknowledgments

Just as games are generally not developed by a single person but by a team, this book was a team effort as well. First of all, I'd like to thank my co-authors Jeroen and Mark for their inspiring ideas and the interesting discussions we've had while writing this book.

This book is based on reading material that was handed out to the (many) students of the game programming course here at Utrecht University. Their feedback and critical analysis of the material have been of great help, as well as their motivation to work with the example games. I officially apologize for any frustration that occurred due to a few ridiculously difficult levels of the Tick Tick game!

I would like to thank my colleagues for their ideas and their interest in this work. In particular, I would like to thank Cathy Ennis and Sybren Stüvel for providing corrections and for reading the text in detail. Their feedback has resulted in many improvements of the text and the sample programs.

The sprites for all the example games in this book were designed by Heiny Reimes. It was a pleasure working with him on designing the example games and improving them.

I would also like to thank Ralf Gerstner from Springer for taking the time to read the manuscript and helping to make this book a reality.

Finally, I would like to thank my wife, Sterre, for her continuing support. I dedicate this work to her.

Utrecht, The Netherlands
January 2013

Arjan Egges

For this second edition, I've been given the chance to improve the book and add my own insights and twists. I should start by thanking everyone who made the *first* edition possible: you gave me a luxurious starting point with logically chosen topics and fun example games. Thanks to Arjan for entrusting me with this update, and thanks again to Ralf Steiner at Springer, who allowed me to write a much bigger overhaul than we'd initially planned.

The improvements in this new edition are based on my teaching experience at Utrecht University. I'd like to thank Paul Bergervoet for guiding me into the Game Programming course there in 2017 and his successor Remco Veltkamp for his interest in improving it with me. Also, thanks to all the students who've provided feedback on the first edition. Your comments have definitely helped me turn this book into a smoother resource for students.

Special shout-out to Jeroen van Knotsenburg, a C# guru with a keen eye for clean and reusable code but (most importantly) a great friend who's been nearby ever since we took our own first programming course together.

Last but not least, I sincerely thank Nicoline, who's not just one of the most intelligent people I've ever met but also a really cool partner.

Rennes, France
January 2019

Wouter van Toll

Contents

Part I Getting Started

1	Building Your First Game Application	3
1.1	Getting and Installing the Tools	3
1.2	Creating Your First Game Application	3
1.3	Running the Game Project	5
1.4	Changing Your First Line of Code	5
1.5	Projects and Solutions	7
1.6	Running the Examples in This Book	7
1.7	What You Have Learned	8
2	What Is Programming?	9
2.1	Computers and Programs	9
2.2	Programming Languages	10
2.2.1	Imperative Programming: Instructions	11
2.2.2	Procedural Programming: Instructions in Methods	11
2.2.3	Object-Oriented Programming: Methods in Classes	12
2.2.4	Java	13
2.2.5	C#	13
2.2.6	Other Modern Languages	14
2.3	Types of Applications	15
2.4	Game Programming and Game Engines	16
2.5	Translating a Program	17
2.5.1	Assembler	17
2.5.2	Compiler	18
2.5.3	Interpreter	18
2.5.4	Compiler and Interpreter	19
2.5.5	Two Compilers	19
2.5.6	A Final Note	20
2.6	Syntax and Semantics	20
2.7	Development Cycles	21
2.7.1	Small Scale: Edit-Compile-Run	21
2.7.2	Large Scale: Design-Specify-Implement	21
2.8	What You Have Learned	22
2.9	Exercises	23

3	Game Programming Basics	25
3.1	Structure of a C# Program	25
3.1.1	Instructions	25
3.1.2	Methods: Grouping Instructions Together	26
3.1.3	Methods and Parameters	26
3.1.4	Classes: Grouping Methods Together	26
3.1.5	A Class Is a Blueprint for Objects	27
3.2	Hello World: The Smallest C# Program You Will Ever See	27
3.3	Building Blocks of a Game	29
3.3.1	The Game World	29
3.3.2	The Game Loop	30
3.3.3	How the Game Loop Gives the Illusion of Movement	31
3.4	Structure of a MonoGame Application	31
3.4.1	The Game Loop in MonoGame	32
3.4.2	Other Methods of the Game Class	33
3.4.3	The Graphics Device	34
3.5	Structure of a Larger Program	35
3.5.1	Namespaces: Grouping Classes Together	35
3.5.2	Libraries: Relying on Other Classes	36
3.5.3	Compilation Units	37
3.6	Program Layout	37
3.6.1	Comments: Explaining What You're Doing	37
3.6.2	Instructions vs. Lines	39
3.6.3	Whitespace and Indentation	39
3.7	What You Have Learned	40
3.8	Exercises	40
4	Creating a Game World	41
4.1	Variables and Types	41
4.1.1	Types	41
4.1.2	Declaring a Variable	42
4.1.3	Assigning a Value to a Variable	43
4.1.4	Declarations and Assignments: More Complex Examples	44
4.1.5	Constants	45
4.2	Expressions: Pieces of Code with a Value	45
4.2.1	Expressions Versus Instructions	46
4.2.2	Operators: Combining Expressions into New Expressions	46
4.2.3	Priority of Operators	47
4.2.4	Variants of Operators	48
4.3	Other Data Types	49
4.3.1	The Double Type	49
4.3.2	Data Types for Integers	50
4.3.3	Data Types for Real Numbers	50
4.3.4	Mixing Different Data Types	51
4.3.5	Data Types in General	52
4.3.6	Classes, Instances, and Data Types	52

4.4	Putting It to Practice: A Changing Background Color	53
4.4.1	Using a Variable to Store a Color	54
4.4.2	Creating Your Own Colors	54
4.4.3	Using the <code>GameTime</code> Class to Change the Color	55
4.4.4	The Scope of a Variable	57
4.4.5	Creating a Member Variable	58
4.4.6	Summary of the <code>DiscoWorld</code> Example	59
4.5	What You Have Learned	61
4.6	Exercises	61

Part II Game Objects and Interaction

Introduction	67
5 Showing What the Player Is Doing	69
5.1 Using Game Assets in <code>MonoGame</code>	69
5.1.1 Managing Assets with the Pipeline Tool	70
5.1.2 Telling <code>MonoGame</code> Where to Load Your Assets	72
5.1.3 Loading Sprites	73
5.1.4 Drawing Sprites	74
5.1.5 Music and Sounds	75
5.2 A Sprite Following the Mouse Pointer	77
5.2.1 Adding a Background	77
5.2.2 Using a <code>Vector2</code> Variable for the Position of a Sprite	78
5.2.3 Retrieving the Mouse Position	80
5.3 Changing the Sprite Origin	82
5.3.1 Using the Width and Height of a Sprite	82
5.3.2 Vector Math	82
5.3.3 Multiple Versions of a Method	84
5.4 A Rotating Cannon Barrel	85
5.4.1 More Sprites and Origins	85
5.4.2 An Angle Variable	86
5.4.3 Computing Angles with the <code>Math</code> Class	86
5.5 What You Have Learned	88
5.6 Exercises	88
6 Reacting to Player Input	91
6.1 Reacting to a Mouse Click	91
6.1.1 Enumerated Types	91
6.1.2 Executing Instructions Depending on a Condition	92
6.2 Boolean Expressions	95
6.2.1 Comparison Operators	95
6.2.2 Logical Operators	96
6.2.3 Boolean Variables	97
6.3 More on <code>if</code> Instructions	98
6.3.1 The <code>else</code> Keyword	98
6.3.2 A Number of Different Alternatives	99
6.3.3 Handling Mouse Clicks Instead of Presses	101
6.4 A Multicolored Cannon	102
6.4.1 Choosing Between Multiple Colors of Sprites	103
6.4.2 Handling Keyboard Input	104

6.5	What You Have Learned	105
6.6	Exercises	106
7	Basic Game Objects	107
7.1	Grouping Instructions into Methods	107
7.1.1	A Method for Input Handling	108
7.1.2	Methods with Parameters	109
7.1.3	The Scope of Method Parameters	110
7.1.4	Methods with a Result	111
7.1.5	More on the return Keyword	112
7.1.6	More on Return Values	113
7.1.7	Expressions, Instructions, and Method Calls	115
7.2	Organizing a Game into Classes	116
7.2.1	Classes, Instances, and Objects	117
7.2.2	Instances and Member Variables	117
7.2.3	Instances and Method Calls	118
7.2.4	Summary: How All Concepts Are Related	119
7.2.5	Access Modifiers	120
7.2.6	this : The Object That Is Being Manipulated	121
7.3	Putting It to Practice: Adding a Cannon Class to Painter	123
7.3.1	Adding a Class File to the Project	123
7.3.2	Creating an Instance of Cannon in the Game	124
7.3.3	Adding Member Variables	125
7.3.4	Adding Your Own Constructor Method	125
7.3.5	Adding a Parameter to the Constructor	126
7.3.6	Adding Other Methods	128
7.4	Accessing the Data in the Cannon Class	129
7.4.1	Option 1: Marking Member Variables as public	130
7.4.2	Option 2: Adding Getter and Setter Methods	130
7.4.3	Option 3: Adding Properties	131
7.4.4	Using the Properties of Cannon in the Painter Class	134
7.4.5	More on Properties	134
7.5	Reorganizing the Class Structure	135
7.5.1	A Class for Handling Input	135
7.5.2	Letting the Cannon Class Maintain Itself	138
7.5.3	A Class for Representing the Game World	139
7.6	What You Have Learned	142
7.7	Exercises	142
8	Communication and Interaction Between Objects	145
8.1	How Objects Are Stored in Memory	145
8.1.1	Primitive Types: Values	145
8.1.2	Class Types: References	146
8.1.3	The Consequence for Method Parameters	147
8.1.4	The null Keyword	149
8.1.5	Reference Counting and Garbage Collection	150
8.1.6	Classes Versus Structs	150
8.1.7	Complex Objects	151

8.2	The Ball Class and Its Interaction with the Game World	151
8.2.1	Structure of the Ball Class	151
8.2.2	Overview of the Ball Behavior	154
8.2.3	static : Making the Game World Accessible Everywhere	154
8.2.4	Drawing the Ball at the Cannon Tip	157
8.2.5	Letting the Ball Move	157
8.2.6	Giving the Ball the Correct Launch Velocity	160
8.2.7	Applying Gravity	160
8.2.8	Resetting the Ball When It Leaves the Screen	161
8.3	The PaintCan Class	162
8.3.1	Structure of the Class	162
8.3.2	Creating Three Different Paint Cans	163
8.3.3	Making the Paint Cans Loop Around	165
8.3.4	Taking the Origin into Account	165
8.3.5	Giving the Cans Random Speeds	166
8.3.6	More Randomness	168
8.4	Handling Collisions Between the Ball and Cans	169
8.4.1	Using Rectangular Bounding Boxes	170
8.4.2	Responding to a Collision	171
8.5	What You Have Learned	171
8.6	Exercises	172
9	A Limited Number of Lives	173
9.1	Maintaining the Number of Lives	173
9.2	Loops: Executing Instructions Multiple Times	174
9.2.1	The while Instruction	174
9.2.2	The for Instruction: A Compact Version of while	176
9.3	Special Cases of Loops	178
9.3.1	Zero Iterations	178
9.3.2	Infinite Loops	178
9.3.3	Nested Loops	179
9.3.4	Loops Inside the Game Loop	180
9.3.5	The Keywords continue and break	181
9.4	Adding a “Game Over” State	182
9.5	More on the Scope of Variables	184
9.5.1	Variable Scope Inside a Loop	184
9.5.2	Conclusions	185
9.6	What You Have Learned	186
9.7	Exercises	186
10	Organizing Game Objects	189
10.1	Introduction to Inheritance	189
10.1.1	Motivation: Similarities Between Game Objects	189
10.1.2	Example and Terminology	190
10.2	Creating Your First Base Class: ThreeColorGameObject	191
10.2.1	Member Variables	191
10.2.2	Methods and Properties	191
10.2.3	Constructor	193

10.3	Turning Cannon into a Subclass	193
10.3.1	Class Outline, Member Variables, and Properties	194
10.3.2	Changing the Constructor	196
10.3.3	protected : Making Things Available to Base Classes	197
10.3.4	Overriding Methods from the Base Class	198
10.3.5	Reusing Base Methods: The base Keyword	200
10.4	Turning Ball and PaintCan into Subclasses	202
10.5	More on Inheritance	202
10.5.1	Polymorphism	203
10.5.2	Hierarchies of Classes	204
10.5.3	Sealed Methods and Classes	206
10.5.4	How Not to Use Inheritance	206
10.6	What You Have Learned	206
10.7	Exercises	207
11	Finishing the Game	211
11.1	Adding the Finishing Touches	211
11.1.1	Motion Effects	211
11.1.2	Sounds and Music	212
11.1.3	Maintaining a Score	212
11.2	Dealing with Text in C#	213
11.2.1	The char Data Type	213
11.2.2	The string Data Type	214
11.2.3	Special Characters and Escaping	215
11.2.4	Operations with Strings	216
11.2.5	Operations with Characters	217
11.3	Putting It to Practice: Drawing the Score	218
11.3.1	Adding a Font in MonoGame	218
11.3.2	Drawing the Score on the Screen	219
11.4	Writing Documentation	219
11.5	What You Have Learned	221
11.6	Exercises	221
 Part III Structures and Patterns		
Introduction		225
12	Dealing with Different Screen Sizes	227
12.1	Scaling the Game World to Fit in the Window	227
12.1.1	Changing the Window Size	228
12.1.2	Storing the World Size and the Window Size Separately	228
12.1.3	Calculating How to Scale the Game World	229
12.1.4	Scaling the Game World	230
12.2	Making the Game Full-Screen	230
12.2.1	Allowing Full-Screen Mode and Windowed Mode	231
12.2.2	Toggling Between the Two Modes	232
12.2.3	Allowing the Player to Quit the Game	233
12.3	Maintaining the Aspect Ratio	233
12.3.1	What Is a Viewport?	233
12.3.2	Calculating the Viewport	234
12.3.3	Using the Viewport for Drawing Sprites	235

12.4	Screen Coordinates and World Coordinates	236
12.4.1	Converting Screen Coordinates to World Coordinates	237
12.4.2	Showing the Mouse Position	237
12.5	What You Have Learned	238
12.6	Exercises	238
13	Arrays and Collections	241
13.1	Arrays: Storing a Sequence of Objects	241
13.1.1	Basic Usage of Arrays	242
13.1.2	Arrays in Memory	243
13.1.3	Combining Arrays with Loops	245
13.1.4	Multidimensional Arrays	245
13.1.5	Arrays of Arrays	247
13.1.6	Shorthand for Initializing an Array	247
13.2	Putting It to Practice: Jewels in a Grid	248
13.2.1	Creating the Grid	248
13.2.2	Filling the Grid with Random Jewels	249
13.2.3	Drawing the Jewels on the Screen	249
13.2.4	Letting the Player Change the Array	251
13.3	Strings and Arrays	252
13.3.1	Immutability	253
13.3.2	Useful String Methods	254
13.3.3	String Operators	254
13.4	Collections	254
13.4.1	The List Class	255
13.4.2	Lists, Loops, and the foreach Keyword	256
13.4.3	Other Collections	257
13.5	What You Have Learned	258
13.6	Exercises	259
14	Game Objects in a Structure	263
14.1	Creating Your Own Game Class	263
14.1.1	Outline, Member Variables, and Properties	263
14.1.2	Methods	264
14.1.3	Turning JewelJam into a Subclass	266
14.2	A List of Game Objects	266
14.2.1	The GameObject Class	267
14.2.2	The SpriteGameObject Class	268
14.2.3	Describing the Game World as a List of Game Objects	269
14.2.4	Collections and Polymorphism	271
14.3	Changing the Grid of Jewels	271
14.3.1	Turning a Jewel into a Game Object	271
14.3.2	Turning the Grid of Jewels into a Game Object	272
14.3.3	Moving Behavior to the JewelGrid Class	274
14.4	A Hierarchy of Game Objects	275
14.4.1	Relations Between Game Objects	276
14.4.2	Global and Local Positions	277
14.4.3	Recursion: A Method or Property Calling Itself	279
14.4.4	The GameObjectList Class	279

14.5	More on Recursion	280
14.5.1	The Two Ingredients of Recursion	281
14.5.2	Example: The Sum from 1 to n	281
14.5.3	Watch Out for Infinite Recursion	283
14.6	What You Have Learned	283
14.7	Exercises	284
15	Gameplay Programming	285
15.1	Selecting Rows and Moving the Jewels	286
15.1.1	Class Overview and Constructor	286
15.1.2	Adding a RowSelector to the Game World	286
15.1.3	Changing the Selected Row	287
15.1.4	Shifting the Jewels in a Row	288
15.2	More Types of Jewels	290
15.2.1	One Sprite Sheet for All Jewels	290
15.2.2	Three Properties of a Jewel	290
15.2.3	Drawing the Correct Part of the Sprite Sheet	291
15.3	Finding Combinations of Jewels	292
15.3.1	Checking if a Combination Is Valid	292
15.3.2	Handling Keyboard Input	293
15.3.3	Removing Jewels from the Grid	294
15.4	Maintaining and Showing the Score	295
15.4.1	A Separate Class for the Game World	295
15.4.2	Updating the Score	297
15.4.3	The TextGameObject Class	297
15.4.4	Showing the Score on the Screen	299
15.4.5	Extra Points for Multiple Combinations	299
15.5	A Moving Jewel Cart	300
15.5.1	The JewelCart Class	300
15.5.2	Adding the Cart to the Game	301
15.6	Multiple Game States	302
15.6.1	Adding Overlay Images	303
15.6.2	Using an Enum to Represent the Game State	304
15.6.3	Different Behaviors per Game State	304
15.6.4	Adding a Help Button	306
15.7	What You Have Learned	308
15.8	Exercises	308
16	Finishing the Game	309
16.1	Making the Jewels Move Smoothly	309
16.1.1	Adding a Target Position	309
16.1.2	Setting the Target Positions of All Jewels	310
16.1.3	Letting the New Jewels Fall from the Sky	311
16.2	Showing Combo Images for a Certain Amount of Time	312
16.2.1	Managing the Visibility of Another Object	312
16.2.2	Adding the Objects to the Game World	313
16.2.3	Applying It to the Combo Images	314

16.3 Adding Glitter Effects	315
16.3.1 Outline of the GlitterField Class	315
16.3.2 Calculating a Random Glitter Position	317
16.3.3 Adding Glitter Fields to Game Objects	318
16.4 Adding Music and Sound Effects	320
16.4.1 An Improved Asset Manager	320
16.4.2 Playing the New Sounds and Music	321
16.5 What You Have Learned	321
16.6 Exercises	322

Part IV Menus and Levels

Introduction	327
17 Better Game State Management	329
17.1 New Classes for Game State Management	329
17.1.1 The GameState Class	330
17.1.2 A Collection of Game States	331
17.1.3 Letting the Active Game State Do Its Job	331
17.1.4 Adding a GameStateManager to the Game	333
17.2 Creating the Game States of Penguin Pairs	333
17.2.1 Adding and Organizing the Assets	333
17.2.2 Adding a Class for Each Game State	333
17.2.3 Preparing the PenguinPairs Class	335
17.2.4 Adding Constants for Object Names	336
17.2.5 Switching Between Game States	337
17.3 Abstract Classes	338
17.3.1 What Is an Abstract Class?	338
17.3.2 Abstract Classes in the Game Engine	339
17.3.3 Abstract Methods	339
17.3.4 Abstract Properties	340
17.4 Interfaces	341
17.4.1 What Is an Interface?	341
17.4.2 Implementing an Interface	342
17.4.3 Implementing Multiple Interfaces	343
17.4.4 The IGameLoopObject Interface	343
17.4.5 Interfaces and Polymorphism	344
17.5 What You Have Learned	346
17.6 Exercises	346
18 User Interfaces and Menus	349
18.1 Sprite Sheets	349
18.1.1 Overview of the SpriteSheet Class	349
18.1.2 Adding a Constructor	351
18.1.3 Reading the Number of Sprites from the Filename	352
18.1.4 A Property for the Sheet Index	353
18.1.5 Updating the SpriteGameObject Class	354

18.2 Menu Elements	354
18.2.1 Obtaining the Mouse Position in World Coordinates	354
18.2.2 The Button Class	355
18.2.3 The Switch Class	356
18.2.4 The Slider Class	357
18.3 Adding the First Menus of Penguin Pairs	360
18.3.1 Filling the Title Screen	360
18.3.2 Adding the Three Back Buttons	360
18.3.3 Adding Buttons to the Playing State	361
18.4 Filling the Options Menu	361
18.4.1 Adding the UI Elements	361
18.4.2 Changing the Music Volume	363
18.4.3 Enabling and Disabling Hints	363
18.5 Filling the Level Selection Screen	364
18.5.1 Defining Level Statuses	364
18.5.2 The LevelButton Class	364
18.5.3 Adding a Grid of Level Buttons	366
18.5.4 Starting a Level	367
18.6 What You Have Learned	368
18.7 Exercises	368
19 Loading Levels from Files	371
19.1 Structure of a Level	371
19.1.1 All Possible Level Elements	372
19.1.2 Defining a File Format	373
19.2 Creating the Classes of Level Elements	374
19.2.1 The Tile Class	374
19.2.2 The Animal Class	376
19.2.3 Subclasses of the Animal Class	376
19.3 Reading and Writing Files	377
19.3.1 Introduction to File I/O	378
19.3.2 I/O in C#: Streams, Readers, and Writers	378
19.3.3 Reading a Text File in C#	380
19.3.4 Reading One Line at a Time	381
19.3.5 Writing a Text File in C#	382
19.4 Putting It to Practice: Reading a Level File	383
19.4.1 Outline of the Level Class	383
19.4.2 Reading the General Data	385
19.4.3 Reading the Lines of the Grid	386
19.4.4 Preparing the Grid	386
19.4.5 Filling the Grid with Tiles and Animals	387
19.4.6 The AddTile Method	388
19.4.7 The AddAnimal Method	389
19.5 Adding Levels to the Game	391
19.5.1 Adding the Level Files to the Project	391
19.5.2 Storing a Level in the Playing State	391
19.5.3 Letting the Playing State Load a Level	392
19.5.4 Loading a Level When the Player Asks for It	393

19.6 Reading and Writing the Player's Progress	393
19.6.1 Representing the Player's Progress	394
19.6.2 Loading the Progress File	394
19.6.3 Saving a New Progress File	395
19.6.4 Showing the Player's Progress in the Level Menu	395
19.7 The switch Instruction: Handling Many Alternatives	396
19.7.1 Basic Usage of switch	396
19.7.2 Using return Instead of break	398
19.7.3 Restrictions	398
19.8 What You Have Learned	399
19.9 Exercises	399
20 Gameplay Programming	401
20.1 Selecting Animals	401
20.1.1 The Arrow Class	401
20.1.2 The MovableAnimalSelector Class: Outline	403
20.1.3 The MovableAnimalSelector Class: Game Loop	404
20.1.4 Handling Clicks on an Animal	405
20.1.5 The Order of Input Handling	406
20.2 Making the Animals Move	407
20.2.1 Linking the Animals to the Grid	407
20.2.2 Overview of the MovableAnimal Design	408
20.2.3 Moving and Stopping	409
20.3 Adding the Game Logic	410
20.3.1 Preparing the Level Class	410
20.3.2 Checking if a Move Is Possible	411
20.3.3 Checking if Two Movable Animals Can Form a Pair	412
20.3.4 Checking if an Animal Is Movable: The Keyword is	413
20.3.5 Applying the New Position of a Movable Animal	414
20.3.6 Cleaning Up the Code	415
20.4 Maintaining the Number of Pairs	415
20.4.1 The PairList Class	416
20.4.2 Storing and Showing Pairs in the Level	417
20.4.3 Adding a Pair at the Right Time	418
20.5 What You Have Learned	418
20.6 Exercises	419
21 Finishing the Game	421
21.1 Hints and Resetting	421
21.1.1 Showing a Hint	421
21.1.2 Resetting a Level	422
21.1.3 Resetting the Other Game Objects	423
21.2 Completing a Level	423
21.2.1 Updating the PlayingState and Level Classes	424
21.2.2 Going to the Next Level	424
21.2.3 Updating the Buttons in the Level Menu	425
21.3 Adding Music and Sound Effects	426

21.4 Using Libraries and Namespaces	426
21.4.1 A Separate Library for the Engine	427
21.4.2 Access Modifiers for Classes	427
21.4.3 A Namespace for the Engine Classes	428
21.5 What You Have Learned	429
21.6 Exercises	430
 Part V Animation and Complexity	
Introduction	433
 22 Creating the Main Game Structure	435
22.1 Changes in the Engine Project	435
22.1.1 Classes for Games with Levels	436
22.1.2 Adding Depth to Game Objects	436
22.1.3 Other Changes	437
22.2 Overview of the Initial Game	438
22.2.1 Main Classes	438
22.2.2 The Tile Class	439
22.2.3 The WaterDrop Class	439
22.2.4 The Goal Object	440
22.2.5 Level Files	440
22.2.6 Loading a Level	441
22.3 Dealing with Large Classes	441
22.3.1 Partial Classes	442
22.3.2 Regions	442
22.4 Exceptions: Dealing with Unexpected Errors	443
22.4.1 Dealing with Exceptions: try and catch	444
22.4.2 Multiple Types of Exceptions	444
22.4.3 Throwing Your Own Exceptions	445
22.4.4 When to Use Exceptions?	446
22.4.5 Exceptions and Casting: The Keyword as	447
22.5 What You Have Learned	448
22.6 Exercises	448
 23 Animated Game Objects	451
23.1 A Class for Animations	451
23.1.1 New Member Variables and Properties	453
23.1.2 Playing an Animation	453
23.2 Adding Animations to Game Objects	454
23.2.1 Managing Multiple Animations	455
23.2.2 Updating the Current Animation	456
23.3 Adding the Player Character to Tick Tick	456
23.3.1 Overview of the Player Class	456
23.3.2 Adding an Instance to the Game	458
23.4 What You Have Learned	458
23.5 Exercises	459

24	Game Physics	461
24.1	Jumping and Falling	461
24.1.1	Applying Gravity	462
24.1.2	Letting the Character Jump	462
24.1.3	Finishing Touches and Design Choices	463
24.2	General Collision Detection	464
24.2.1	Collision Detection with Bounding Volumes	464
24.2.2	Limitations of Bounding Volumes	466
24.3	Adding Collision Detection to Tick Tick	467
24.3.1	Communication Between the Character and the Level	467
24.3.2	Giving the Character a Custom Bounding Box	468
24.3.3	Adding the HandleTileCollisions Method	468
24.3.4	Finding the Range of Tiles to Check	469
24.3.5	Dealing with Level Boundaries	470
24.3.6	Collision Detection with a Single Tile	471
24.3.7	Responding to a Collision with a Tile	472
24.3.8	Dealing with False Positives	473
24.4	Adding Slippery Ice and Friction	474
24.4.1	Detecting the Type of Surface Below the Player	474
24.4.2	Adding Friction to the Game Physics	475
24.5	Pixel-Precise Collision Detection	476
24.5.1	Getting and Storing Transparency Information	477
24.5.2	Checking for Overlap Between SpriteGameObject Instances	477
24.6	What You Have Learned	478
24.7	Exercises	479
25	Intelligent Enemies	481
25.1	A Simple Moving Enemy	481
25.1.1	Preparing the Level and Player Classes	481
25.1.2	Overview of the Rocket Class	482
25.2	Enemies with Timers	484
25.2.1	The Turtle Class	484
25.2.2	The Sparky Class	486
25.3	Patrolling Enemies	486
25.3.1	The Basic PatrollingEnemy Class	487
25.3.2	Subclasses for Variants of the Enemy	487
25.3.3	Adding the Enemies to the Game World	489
25.4	What You Have Learned	490
25.5	Exercises	490
26	Finishing the Game	491
26.1	Level Progression	491
26.1.1	Collecting Water Drops	491
26.1.2	Checking If the Level Has Been Completed	492
26.1.3	Celebrating a Victory	492
26.2	Life and Death	493
26.2.1	Changing the Player Class	493
26.2.2	Changing Other Classes	494
26.2.3	Resetting the Level	494

26.3 Adding a Timer	495
26.3.1 The BombTimer Class: Maintaining and Showing the Time	495
26.3.2 Integrating the Timer into the Game	497
26.3.3 Letting Hot Tiles Speed Up the Timer	497
26.4 Finishing Touches	498
26.4.1 Adding Mountains in the Background	498
26.4.2 Adding Moving Clouds	498
26.4.3 Adding Sound Effects	499
26.5 What You Have Learned	500
26.6 Exercises	501
26.7 What's Next?	503
Glossary	505
Index	509

Part I

Getting Started

Chapter 1

Building Your First Game Application



In this book, you're going to learn how to make your own computer games. After you have finished reading this book, you will be able to make games for different platforms that are ready to be published. As you will see, building games can be as much fun as playing them (or even more!).

At the same time, this book will teach you the basics of one of the most popular programming paradigms: *object-oriented programming*. With the skills acquired via this book, you will be able to create not only professional-looking games but also other kinds of applications. In other words, you'll also learn how to develop software in general, but we will use games as our running examples.

Before you can start making your own games, you need to make sure that you have all the tools available in order to get started. In this chapter, you're going to walk step-by-step through the process of transforming your computer into a game development machine. You will run an example project and make your first small change to its source code.

1.1 Getting and Installing the Tools

In order to develop computer games, you need to install a few tools on your computer. The main tool that you're going to need is the Microsoft Visual Studio environment, in combination with the MonoGame platform. On the accompanying website of this book, you can find detailed instructions on how to obtain and install these tools. The Visual Studio development environment and the MonoGame platform are freely available. Make sure to first install Visual Studio and then MonoGame and not the other way around.

Once you have installed the tools according to the guidelines on the website of this book, run Visual Studio. When it has launched, you will see a screen similar to Fig. 1.1.

1.2 Creating Your First Game Application

Now that you've installed the tools, let's see if everything is working the way it should. To test this, you are going to create a **project**. A project is the basic working environment for creating a game. First, start up the Visual Studio development environment. Once Visual Studio has launched, choose File → New → Project in the main menu. On the left side of the screen that pops up, choose the category MonoGame. Your pop-up screen should then look something like depicted in Fig. 1.2. Now select the

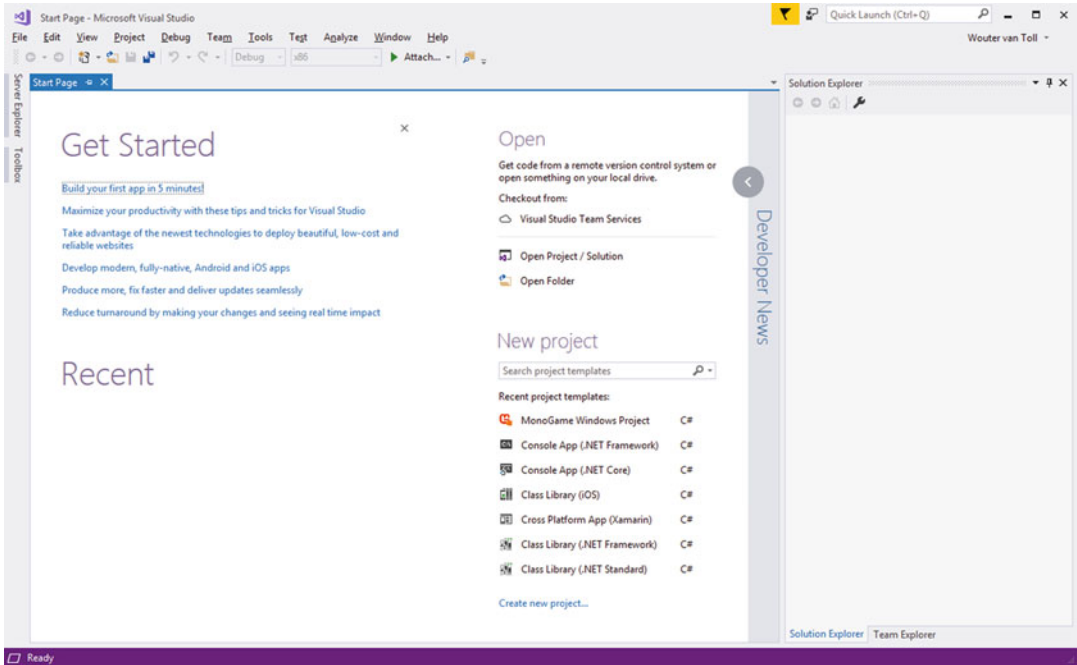


Fig. 1.1 A screenshot of the Visual Studio Community 2017 development environment

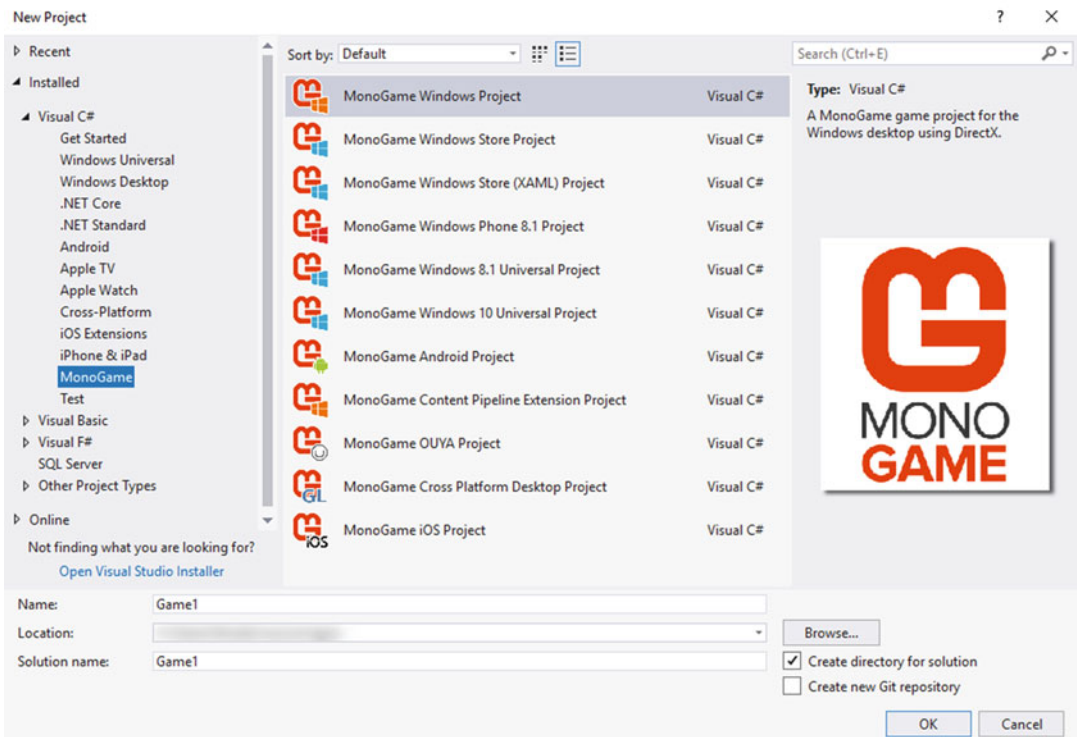


Fig. 1.2 Creating a new MonoGame Windows project



Fig. 1.3 Running the game project

“MonoGame Windows Project” template. At the bottom of the screen, you can enter a name and a location for your project, as well as a few other options that are not important for now. After you’ve chosen a name and a location, click on the OK button. A new project will now be created for you.

1.3 Running the Game Project

Now that you’ve created the project, you can run it by clicking on the Start button (with the green “play” arrow) located at the top middle of the window. Instead of clicking on that button, you can also press F5. The program will then do some work, and eventually you should see the window shown in Fig. 1.3. This window shows the game that you’ve just compiled.

Obviously, this program is not really a game yet, since you cannot do anything except look at the background color.

1.4 Changing Your First Line of Code

Let’s dig a bit deeper into this program: we’re going to look at the *source code* of the game and make our first small change to it.

Close the game window, and take a look at the files in the game project you’ve created. On the right-hand side of the Visual Studio editor window, you’ll see a panel called *Solution Explorer*, which contains a tree structure of files (see Fig. 1.4). The project you created is shown in boldface, and you can see a number of files below it. Double-click on the file *Game1.cs*. You’ll see that the file is a text file that you can edit. This file contains all the code to run the simple program that you’ve just executed.

In the code editor, look for the following line of code (almost at the bottom of the program):

```
GraphicsDevice.Clear(Color.CornflowerBlue);
```

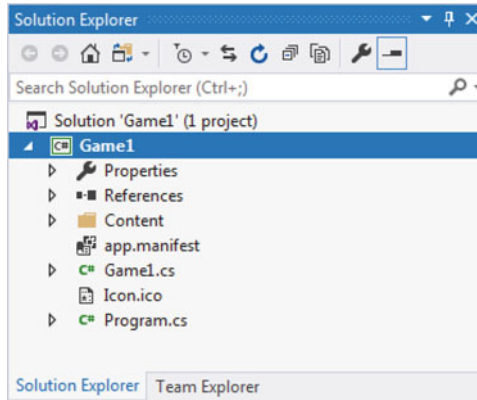


Fig. 1.4 An overview of the files in a basic MonoGame Windows Project

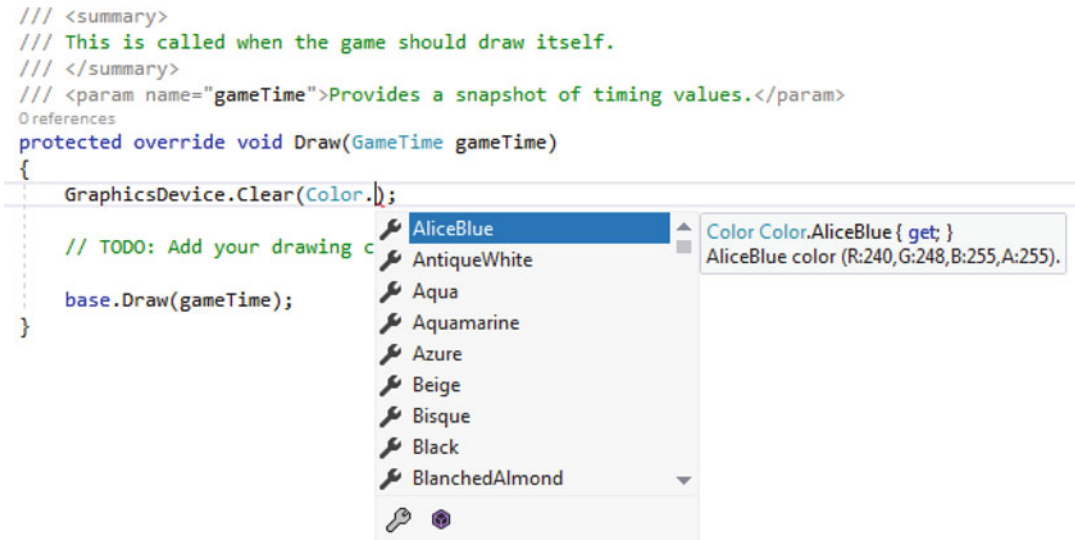


Fig. 1.5 Changing the background color of the main game window

Now, change this line into the following line:

```
GraphicsDevice.Clear(Color.Chocolate);
```

Run the program again by pressing the green play button. The color in the window should now have changed. Instead of Chocolate, you can also write a lot of other color names, such as AntiqueWhite, Olive, Black, and many more. Remove the text `.Chocolate` from the line and enter a dot after the word `Color`, but before the closing parenthesis. You now see that the editor shows a list of all possible colors (see also Fig. 1.5). This is a very nice feature of the editor that works not only for colors but for a lot of other aspects of creating programs. You will use it a lot throughout this book.

1.5 Projects and Solutions

Visual Studio organizes all the games or other programs you develop into **projects** and **solutions**. Each game or application you develop is called a *project*. Projects, in turn, are organized in *solutions*. One solution may contain several projects, where each project represents a game or a program that you're working on.

When you create a new project, you have the possibility to indicate whether you want to create a new solution as well or whether you want to add the new project to the current solution. Having multiple projects in one solution can be very helpful because this makes it easier to work on different projects at the same time. In the *Solution Explorer*, you can see all the projects that are currently in the solution (see Fig. 1.4).

1.6 Running the Examples in This Book

From the book's website, you can download all of the example programs belonging to this book. (Again, please look at the book's website for up-to-date instructions on how to download these example programs.)

Our example programs are all Visual Studio projects, organized in one solution per chapter of the book. For example, to browse through all the samples related to Chap. 5, go to the folder *05_ShowingPlayer* and double-click on the file *ShowingPlayer.sln*. When you look at the Solution Explorer, you'll see that this solution contains various projects.

The project name that is currently in boldface is the project that will be executed when you press the green play button. This project is called the *startup project*. You can select another startup project by right-clicking the project name in the solution explorer and selecting *Set as StartUp Project*. For example, right-click on the *Balloon2* project and set it as the startup project, and then press F5. You should see a window in which the mouse pointer has a balloon attached to it.

In that same solution, you can also find the *Painter1* program, which is the first version of a game that you will develop in Part II of this book: *Painter*. If you open the solution belonging to Chap. 11, you can see that it contains the project *PainterFinal*, which is the final version of that game. Press F5 to create and play the game. Figure 1.6 shows a screenshot of this game.



Fig. 1.6 A screenshot of the first game you're going to create

If you can indeed run the Painter game by pressing F5, this means that everything has been installed correctly. If something isn't working, please have a look at the website of this book for troubleshooting information.

By the way, the goal of the Painter game is to make sure that the paint cans falling down from the top of the screen are in the right color (red, green, or blue) before they fall through the bottom of the screen. You can shoot at the paint cans using the paint cannon in the lower left corner of the screen, and the color to shoot with changes by pressing the R (red), G (green), or B (blue) keys. Aiming and shooting is done by moving the mouse and pressing the left mouse button.

1.7 What You Have Learned

In this chapter, you have learned:

- how to install the Visual Studio development environment and the MonoGame platform;
- how to create a new game project and how to compile and run it;
- how to make your first small change to the source code of a game.

Chapter 2

What Is Programming?



This chapter gives you an introduction to programming in general. We'll first talk about processors and memory, the core elements of any computer. Next, we'll discuss what types of programming languages exist, with a focus on game programming. Then we explain how a program is translated into a format that the computer can efficiently execute. We finish the chapter by briefly discussing the process of developing a program such as a game.

This chapter is very different from the rest of the book. It does not teach you any C# programming concepts yet. Instead, it gives you an insight into what a program actually is, what's happening "in the background" when a program (such as a game) is running, and what the work of a programmer typically looks like. This will give you a better idea of how to write your own programs later on.

If you want to start creating games quickly, you can read Sects. 2.1 and 2.7 and skip the rest of the chapter. It's better to study the entire chapter, though.

2.1 Computers and Programs

Generally speaking, a computer consists of a processor and memory. This is true for all modern computers, including game consoles, smartphones, and tablets.

We define **memory** as something that you can *read* things from and/or *write* things to. An intuitive example of memory is the hard drive on your computer that stores all your files. Memory comes in different varieties, mainly differing in the speed of data transfer and data access. Some memory can be read and written as many times as you want, some memory can only be read, and other memory can only be written to.

You could say that memory represents the current "state" of your computer. By contrast, a **processor** is a part of a computer that does the actual work, which (as a result) changes the computer's memory. The main processor in a computer is called the central processing unit (CPU). The most common other processor on a computer is a graphics processing unit (GPU). Nowadays, even the CPU itself is no longer a single processor, but it often consists of multiple cores.

Input and output equipment (such as a mouse, game controller, keyboard, monitor, printer, touch screen, and so on) seems to fall outside the processor and memory categories at first glance. However, abstractly speaking, they're actually memory. A touch screen is read-only memory, and a printer is write-only memory.

The main task of the processor is to execute **instructions**. The effect of executing these instructions is that the memory is changed. Especially with the very broad definition of memory that we've just given, every instruction changes the memory in some way.

Usually, you probably don't want the computer to execute only one instruction. Generally, you will have a very long list of instructions to be executed—"Move this part of the memory over there, clear this part of the memory, draw this sprite on the screen, check if the player is pressing a key on the game controller, . . ." As you probably expect, such a list of instructions for the processor is called a *program*.

In summary, a **program** is a long list of instructions to change the computer's memory. However, the program itself is also stored in memory. Before the instructions in the program are executed, they're stored on a hard disk, on a DVD, on a USB flash disk, in the cloud, or on any other storage medium. When they need to be executed, the program is moved to the internal memory of the machine.

Programming is the activity of writing programs, and you will learn to do it yourself in this book. Programming is certainly not an easy task, and it requires logical thinking: you need to be constantly aware of what your program will do with the memory later on.

Programs in Real Life? — Let's get a bit philosophical for a moment. If you are willing to broaden your definition of memory and processor, then the concept of a program is no longer specifically meant for computers. Many programs are written and executed in everyday life. A nice example of a program is a recipe for making an apple pie. The values in the memory in this case are the apples, flour, butter, cinnamon, and all the other ingredients. The cook (or the processor) then executes all the instructions in the recipe which changes the values in the memory. After that, we have another type of processor: the person who eats the pie and digests it. Other examples of daily-life programs include following directions for getting to a particular place, logistics strategies for supplying supermarkets, administrative procedures, and so on. All these things have in common that they are long sequences of instructions that—when executed—have a certain effect.

2.2 Programming Languages

The instructions that form a program need to be expressed in some way. Because a computer can't grasp instructions typed in plain English, you need to write your instructions in special programming languages such as C#. In practice, you will write the instructions as text using a normal keyboard, but you need to follow a very strict way of writing them down, according to a set of rules that defines a **programming language**.

Many programming languages exist, because a new programming language is often created whenever somebody thinks of a slightly better way of expressing a certain instruction. It's difficult to say how many programming languages there are in total (because that depends on whether you count all the versions and dialects of a language) but suffice to say that there are at least thousands.

Fortunately, you don't have to learn all these different languages because they have many similarities. In the early days, when computers were just new, the main goal of programming languages was to use the (limited) possibilities of computers to the fullest. Nowadays, computers have incredibly fast processors and huge amounts of memory; along with this, programming languages have become

more advanced, and programs have become much bigger and more complex. Recent languages focus on bringing some order to the chaos that writing modern programs can cause.

Programming languages that share similar properties are said to belong to the same **programming paradigm**. A paradigm refers to a set of practices that is commonly used. We will now discuss the most important programming paradigms, by quickly going through the *history* of programming languages.

2.2.1 Imperative Programming: Instructions

A large group of programming languages belongs to the so-called **imperative programming** paradigm. Imperative languages are based on instructions to change the computer's memory. As such, they're well suited to the processor-memory model described in the previous section. C# is an example of an imperative language.

In the early days, programming computer games was a very difficult task that required great skill. A game console like the popular Atari VCS had only 128 bytes of RAM (random access memory) and could use cartridges with at most 4096 bytes of ROM (read-only memory) that had to contain both the program and the game data. This limited the possibilities considerably. For example, most games had a symmetric level design because that halved the memory requirements for describing a level. The machines were also extremely slow.

Programming such games was done in an *Assembler* language. Assembler languages were the first imperative programming languages. Each type of processor had its own set of Assembler instructions, so these Assembler languages were different for each processor. Because such a limited amount of memory was available, game programmers were experts at squeezing out the last bits of memory and performing extremely clever hacks to increase efficiency. The final programs, though, were unreadable and couldn't be understood by anyone but the original programmer. Fortunately that wasn't a problem, because back then, games were typically developed by a single person.

A bigger issue was that every time a new processor came around (with its own new version of an Assembler language), all the existing programs had to be completely rewritten for that processor. Therefore, a need arose for processor-independent programming languages. This resulted in languages such as *Fortran* (FORMula TRANslation) and *BASIC* (Beginners' All-purpose Symbolic Instruction Code). BASIC was very popular in the 1970s because it came with early personal computers such as the Apple II in 1978, the IBM-PC in 1979, and their descendants. Unfortunately, this language was never standardized, so every computer brand used its own dialect of BASIC.

2.2.2 Procedural Programming: Instructions in Methods

As programs became more complex, it was clear that a better way of organizing all these instructions was necessary. In the **procedural programming** paradigm, related instructions are grouped together in *procedures* (also called *functions* or **methods**, depending on the programming language). Because a procedural programming language still contains instructions, all procedural languages are also imperative.

The first real procedural language was *Algol* (an abbreviation of Algorithmic Language). This language was launched in 1960, together with an official definition of the language, which was crucial for exchanging programs between computers or programmers. To describe the structure of Algol programs, a special notation called the Backus Normal Form (BNF) was created. Near the end of the 1960s, a new version of Algol came out, called *Algol68* (guess in which year). Algol68 contained

a lot of new features, as opposed to the original Algol language. In fact, it contained so many new things that it was very complicated to build good Algol68 translators. As a result, only a few of these translator programs were built, and it was soon “game over” for Algol68.

In 1971, a programming language much simpler than Algol68 was created: *Pascal* (not an abbreviation this time but a reference to the French mathematician Blaise Pascal). Pascal was created by Niklaus Wirth, a professor at the university of Zürich, with the goal to provide students with an easy-to-learn programming language. Soon, this language was also used for more serious applications.

For really big projects, Pascal was not suitable, though. An example of such a big project was the development of the Unix operating system at the end of the 1970s at Bell Labs. Because an operating system is a very complicated kind of program, Bell Labs wanted to write it in a procedural language. The company defined a new language called *C* (because it was a successor of earlier prototypes called A and B). The philosophy of Unix was that everybody could write their own extensions to the operating system, and it made sense to write these extensions in C as well. As a result, C became the most important procedural language of the 1980s, also outside the Unix world.

Today, C is still being used quite a lot, although it’s gradually making way for more modern languages, especially in the game industry. Over the years, games have become much larger programs, they are now often created by large teams rather than individuals, and it has become more important that programmers spend their time efficiently. As such, it gradually became more important for game code to be readable, reusable, and easy to debug. Although C was already a lot better in that respect than the Assembler languages, it remained difficult to write very large programs in a structured way.

2.2.3 Object-Oriented Programming: Methods in Classes

Procedural languages like C allow you to group instructions in procedures (also called methods). Just as they realized that instructions belonged together in groups, programmers saw that some methods belonged together as well. The **object-oriented programming** paradigm lets programmers group methods into something called a **class**. The memory that these groups of methods can change is called an **object**. For example, a class can describe the behavior of the ghosts in a game of Pac-Man. Then each individual ghost corresponds to an object of that class. This way of thinking about programming is powerful when applied to games: roughly speaking, many “objects” that you see in a game are also object in the underlying program.

The first object-oriented language ever was *Simula*, created in 1967 by the Norwegian researchers Ole-Johan Dahl and Kristen Nygaard. They developed Simula as an extension of Algol, because they were interested in doing simulations to analyze traffic flux or the behavior of queues of people in the post office. One of the new things in Simula was that it identified a group of variables in the memory as an *object*. This was very useful, because it was intuitive to describe (for example) a person or a car as an object.

Simula itself was not a very popular language, but the idea to describe a program in terms of objects and classes was picked up by researchers at Xerox in Palo Alto, who—even before Apple and Microsoft—were experimenting with window-based systems and an actual mouse. Their language called *Smalltalk* used objects to model windows, buttons, and scroll bars: all more or less independent objects. But Smalltalk was carrying things to the extreme: absolutely everything was supposed to be an object, and as a result the language was not very easy to use.

At some point, everybody was already programming in C, so it made sense to create an object-oriented language that was very much like C, except that it let programmers use classes and objects. This language was created in the form of **C++** (the two plus signs indicated that this language was a successor to C). The first version of C++ dates from 1978, and the official standard appeared in 1981.

Although the language C++ is standard, C++ doesn't contain a standard way to write window-based programs on different types of operating systems. Writing such a program on an Apple computer, a Windows computer, or a Unix computer is a completely different task, which makes running C++ programs on different operating systems a complicated issue. Initially, this wasn't considered a problem. However, as the Internet became more popular, the ability to run the same program on different operating systems was increasingly convenient.

2.2.4 Java

The time was ripe for a new programming language: one that would be standardized for usage on different operating systems. The language needed to be similar to C++, but it was also a nice opportunity to remove some old-fashioned C elements to simplify things. The language **Java** fulfilled this role (Java is an Indonesian island famous for its coffee). Java was launched in 1995 by the hardware manufacturer Sun, which used a revolutionary business model for that time: the software was free, and the company planned to make money via support. Also important for Sun was the need to compete with the growing popularity of Microsoft software, which didn't run on the Unix computers produced by Sun.

One of the novelties of Java was that the language was designed so programs couldn't accidentally interfere with other programs running on the same computer. In C++, this was becoming a significant problem: if such an error occurred, it could crash the entire computer or worse—evil programmers could introduce viruses and spyware. Therefore, Java works in a fundamentally different way than C++. Instead of executing instructions directly on the processor, there is another program called a *virtual machine* that controls the instructions and that checks that memory is used only as it is indicated in the program.

2.2.5 C#

In the meantime, Microsoft was also working on their own object-oriented language called **C#**, which was launched in 2000. C# is an object-oriented language that—like Java—uses a virtual machine (which Microsoft calls *managed code*). The name of the language already indicates that it continues in the tradition of C and C++. Typographically, the hash sign even resembles four plus signs. In musical notation, the hash sign symbolizes a sharp note, and the language is therefore pronounced as “C-Sharp.” A nice detail is that “sharp” also means “smart.” So: C# is a smarter version of C.

We have chosen for C# as the programming language of this book. C# is much easier to learn than C++, which is complicated due to its compatibility with C. By learning C#, you automatically also learn many basic object-oriented programming concepts, which is a very useful skill to have. And if you ever need to write a program in C++ or Java, you can reuse your knowledge of C# because of the many similarities between these languages.

Furthermore, C# is used in many different ways in the (game) industry. For instance, the Unity3D game engine uses C# as its main scripting language (next to JavaScript). The game engine *MonoGame*, used in this book, is also written in C#. We will discuss game engines a bit more in Sect. 2.4.

Finally, C# is also used by tools such as Xamarin that allow you to create apps for iOS and Android with a single programming language. This is more convenient than having to learn separate programming languages for both systems (namely, Java for Android and Swift or Objective-C for iOS).

In short, C# is a modern language that is used a lot for game development (and software development in general), and it is very suitable as a first programming language to learn.

2.2.6 Other Modern Languages

Obviously, the world hasn't stopped since C# was created. A number of new (and improved old) languages have appeared. One example of such a language is *JavaScript*, a language invented by Netscape as a means to dynamically change the contents of a website. JavaScript combines elements from both the imperative and declarative programming paradigms. Because it is well suited for client-server communication, JavaScript is used everywhere nowadays. Many companies use JavaScript and HTML to define the user interface of their applications. With powerful libraries such as AngularJS and Ionic, JavaScript has become very popular for (mobile) application development in general. You can even develop games in JavaScript.

Another language that should be mentioned in this section is *Swift*, the language created by Apple to replace Objective-C. Like JavaScript, Swift also combines elements from various programming paradigms. It builds upon features introduced in other languages (such as C#) and it introduces new features itself, such as optionals (a very nice way of explicitly dealing with variables that may or may not refer to an object).

In the end, many programming languages are not really that different from each other. It doesn't really matter which programming language you learn as a starting point. Once you've learned one language, you can learn a second or a third language relatively quickly. On the long term, the main challenge lies in thoroughly grasping the major programming concepts, such as properly dividing a program into objects to keep your code clean, knowing when to use which data structures and algorithms to get the best performance, and using common design patterns such as the MVC (model-view-controller) framework.

In this book, we assume that you have not yet programmed (much) before, so we will focus on the basics of object-oriented programming. Data structures, algorithms, and design patterns are more advanced concepts: there are often separate courses for this in a computer science education program. It is useful to obtain a solid basis in programming first.

Declarative Programming — Now that you know about the imperative programming paradigm, you might wonder if there are any other programming paradigms that are *not* based on instructions. Is this possible? What does the processor do if it doesn't execute instructions?

Well, the processor always executes instructions, but that doesn't mean the programming language has to contain them. For example, suppose you build a very complicated Excel spreadsheet with many links between different cells in the sheet. You could refer to this activity as programming, and you could call the empty spreadsheet the program, ready to process data. This type of programming is called **functional programming**. In a functional language, a program is entirely written as functions that somehow depend on each other. A popular functional programming language is *Haskell*.

There are also languages based on logic, the so-called **logical programming** languages. An example of such a language is *Prolog*. A Prolog program consists of logical formulas that describe how properties lead to other properties. For instance, a Prolog program could contain the rule "every cat has a tail," along with the knowledge that "Bob is a cat." (We ignore the

(continued)

official language rules of Prolog in this example.) If you ask this program the question “does Bob have a tail?”, the program will respond with “yes,” because it can *infer* this fact from its knowledge and logical rules.

These two types of programming languages (functional and logical) form the **declarative programming** paradigm. But let’s not worry about these languages in this book and focus on C# from now on.

2.3 Types of Applications

A computer program with which users can interact is often called an **application**. Let’s quickly review what kinds of applications exist.

In the early days, many computer programs only wrote text to the screen and didn’t use graphics. Such a text-based application is called a *console* application. In addition to printing text to the screen, these applications could also read text that a user entered on the keyboard. So, any communication with the user was done in the form of question/answer sequences (*Do you want to format the hard drive (Y/N)?*, *Are you sure (Y/N)?*, and so on). Before Windows-based operating systems became popular, this text-based interface was very common for text-editing programs, spreadsheets, math applications, and even games. These games were called text-based adventures, and they described the game world in text form. The player could then enter commands to interact with the game world, such as *go west* or *pick up matches*. Examples of such early games are *Zork* and *Adventure*. Although they might seem dated now, they’re still fun to play!

It’s still possible to write console applications in a modern language like C#. In fact, you’ll see such an application (called “HelloWorld”) in the next chapter. This book will focus on programming games with graphics, but sometimes console applications are useful to explain a programming concept in the shortest possible way.

Nowadays, many other types of applications exist, including the following:

- A *windowed application* shows a screen containing windows, buttons, and other parts of a *graphical user interface* (GUI). This type of application is often *event-driven*: it reacts to events such as clicking a button or selecting a menu item.
- In a *web application*, the program is stored on a server and the user runs the program in a web browser. There are many examples of such applications: think of web-based email programs or social network sites. Web applications generally rely on a combination of code on the server (e.g., to access a database) and code on the client side (to show the user interface and handle user interaction). An *app* is an application that runs on a mobile phone or a tablet. Such devices have small screens compared to PC monitor, but they also allow new types of interaction. For example, a mobile device has GPS (to find out the location of the device), sensors that detect the orientation of the device, and a touch screen.
- A *game* is a graphical application, but with fast-moving images. It can be run on a PC, on specialized hardware such as a game console, or on a mobile device—in the last case, the game is also a type of app. Games often use a controller made specifically for the hardware, or the touch screen in the case of game apps.

Not all programs fall squarely in one application type. Some Windows applications might have a console component, such as the JavaScript console in a browser. Games often also have a window

component, such as an inventory screen, a configuration menu, and so on. And nowadays the limit of what a program actually is has become less clear. Think about a multiplayer game that has tens of thousands of players, each running an app on a tablet or an application on a desktop computer, while these programs communicate with a complex program running simultaneously on many servers. What constitutes the program in this case? And what type of program is it?

2.4 Game Programming and Game Engines

Computer games are very interesting programs, also from a programmer's perspective. Games deal with a lot of different input and output devices, they need to respond quickly to the player's input, they need to draw new images on the screen at many frames per second, and the imaginary worlds that games create can be extremely complex. As a result, *game programming* is a category of programming with special properties and requirements.

Until the beginning of the 1990s, games were developed for specific platforms. For example, a game written for a particular console couldn't be used on any other device without major effort from the programmers to adapt the game program to the differing hardware. For PC games, this effect was even worse. Nowadays, operating systems provide a so-called *hardware abstraction layer*, so that programs don't have to deal with all the different types of hardware that can be inside a computer. Before that existed, each game needed to provide its own drivers for each graphics card and sound card; as a result, not much code written for a particular game could be reused for another game. In the 1980s, arcade games were extremely popular, but almost none of the code written for them could be reused for newer games because of the constant changes and improvements in computer hardware.

As games grew more complex, and as operating systems became more hardware independent, it made sense for the game companies to start reusing code from earlier games. Why write an entirely new rendering program or collision-checking program for each game if you can simply use the one from your previously released game?

The term "game engine" was coined in the 1990s, when first-person shooters such as Doom and Quake became a very popular genre. These games were so popular that their manufacturer, id Software, decided to license part of the game code to other game companies as a separate piece of software. Reselling the core game code as a game engine was lucrative: other companies were willing to pay a lot of money for a license to use the engine for their own games. These companies no longer had to write their own game code from scratch—they could reuse the programs contained in the game engine and focus more on graphical models, characters, levels, and so on.

In general, a **game engine** is a set of tools meant to make game development substantially easier. Advanced game engines provide a lot of commonly required functionality to game developers, such as 2D and 3D rendering, special effects such as particles and lighting, sound, animation, artificial intelligence, scripting, and much more. Sometimes, a game engine is also called *game middleware*, because it forms a layer between the basic programming language and the actual game code.

Many different game engines are available today. Some of them are built specifically for a certain game console or operating system. Others can be used for different platforms without having to change the code, such that a single game can easily be released on multiple platforms. *Unreal Engine* and *Unity3D* are two popular examples of such game engines.

Because game engines make programming so much easier, they give game companies more time for creating beautiful environments and challenging levels. Many game development teams contain more artists and designers than programmers. However, programmers are still necessary for writing everything that isn't already included in the engine. Some companies even create and maintain their own (version of a) game engine, which of course also requires programmers.

In this book, we will use **MonoGame**, a game engine written in the C# programming language. With MonoGame, you can create games for a wide variety of platforms such as Windows, Mac, iOS, or Android. Although this book focuses on developing 2D games, you can also develop 3D games in MonoGame. MonoGame provides tools for things you commonly need in games, such as loading and displaying images (sprites), playing sounds, handling collisions between objects in a game, and much more. This book introduces the most important tools and shows you how to use them to create your own games.

MonoGame and XNA — MonoGame is a platform-independent, rewritten version of the XNA game engine. XNA was initially developed by Microsoft for the XBox game console, but it was unfortunately abandoned in 2013. Its successor MonoGame is an open-source project that is being maintained by a worldwide community. It is suitable for creating games that can be played on PCs, mobile devices, and consoles.

The creators of MonoGame made their engine in such a way that old XNA code would still work. In the games of this book, you'll often see lines such as this one:

```
using Microsoft.Xna.Framework;
```

This was a deliberate choice from the MonoGame programmers. So, even though Microsoft is not officially involved anymore, some parts of the engine still hint to MonoGame's history as a Microsoft product.

2.5 Translating a Program

By now, you know that modern programming languages and game engines have made (game) programming much easier than in the early days. Thanks to programming languages, you don't have to write machine-specific instructions yourself. However, the programs you write still need to be translated to a format that computers can immediately understand. This translation is done by a separate special program that (luckily!) has been created for us. In this section, we briefly explain how this translation works.

Depending on the circumstances, the program that translates your code to "real" computer instructions is called an *assembler*, a *compiler*, or an *interpreter*. Let's look at the differences between these three types.

2.5.1 Assembler

An **assembler** is used to translate Assembler programs to instructions that the processor can execute (also called *machine code*). Because an Assembler program is different for each processor, you need different programs for different computers, each of them being translated by the corresponding assembler (see Fig. 2.1).

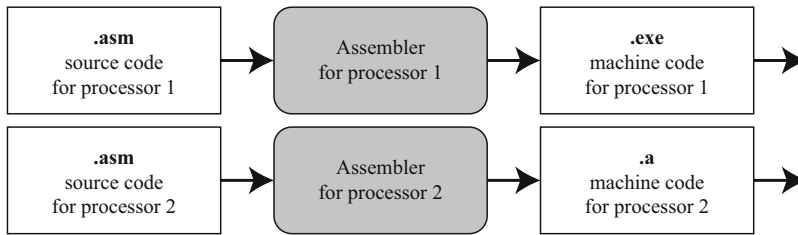


Fig. 2.1 Translating a program using an assembler

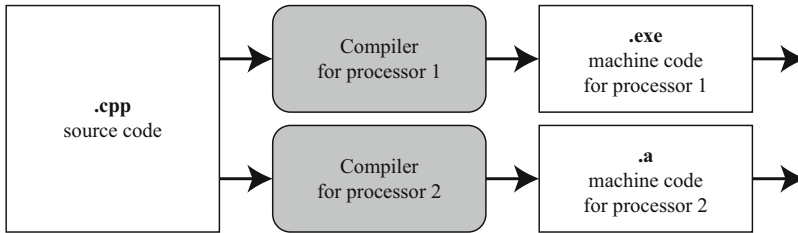


Fig. 2.2 Translating a program using a compiler

2.5.2 Compiler

In many programming languages (except Assembler languages), you can write your program independently of the computer that the program will be run on. A **compiler** is a program that automatically translates your code so that it can be executed on a particular type of machine. For example, if you compile your program for Windows machines, your compiler will typically create a file with the extension *.exe*. To let your program run on different types of machines, you can use different compilers.

The difference between an assembler and a compiler is this: an assembler is specifically made to translate a processor-dependent program to machine code, but a compiler can translate a program written in a processor-independent language to machine code. The compiler itself is machine-specific, because it targets the machine code of the computer that the program has to run on. The program written by the programmer (also called the *source code*) is machine-independent. Many procedural languages, such as C and C++, use a compiler to translate the program into machine code. Figure 2.2 shows what the translation process looks like when using a compiler.

2.5.3 Interpreter

A more direct way to translate programs is to use an **interpreter**. An interpreter reads the source code and *immediately executes* the instructions contained within it, without translating it to machine code first. The interpreter is specific for a machine, but the source code is machine-independent, just like with the compiler.

In a way, an interpreter program is comparable to a human interpreter at a meeting: it translates and speaks at the same time. By contrast, a compiler is comparable to a human translator who converts a written text into another text and eventually shows the result.

The advantage of an interpreter over a compiler is that translating into machine code is no longer required. The disadvantage is that the execution of the program is slower, because translating a pro-

gram on-the-fly takes time. Furthermore, any errors in the program cannot be detected beforehand. Translating using an interpreter is commonly used for script languages such as PHP (see Fig. 2.3).

2.5.4 Compiler and Interpreter

Another approach is to first use a compiler and then an interpreter. The Java programming language uses this approach. First, a compiler translates the source code—but this time, the result is not machine code but a machine-independent intermediate product, called the *bytecode* (see Fig. 2.4). The bytecode can be sent to users, and it will be executed on the user's machine by an interpreter.

Because the main translation is already done by a compiler, interpreting the bytecode is relatively fast. However, a program compiled directly to machine code will always run faster.

2.5.5 Two Compilers

When you write a C# application in Visual Studio (Microsoft's development environment), you use yet another variation. The Microsoft C# compiler generates code in an *intermediate language* similar to the bytecode in Java. This intermediate language can be generated from multiple different programming languages. As a result, bigger projects can integrate programs written in different programming languages. The intermediate language is not interpreted, but it is translated again into machine code for a specific platform by another compiler (see Fig. 2.5).

The file containing the intermediate code is called an *assembly*. However, this has nothing to do with the Assembler languages we've discussed earlier in this chapter.

Sometimes, the compilation into machine code happens at a very late stage, when the program is already running and it needs a part that is not yet compiled. In this approach, the difference between a compiler and an interpreter is not so clear anymore. Because the compiler compiles parts of the

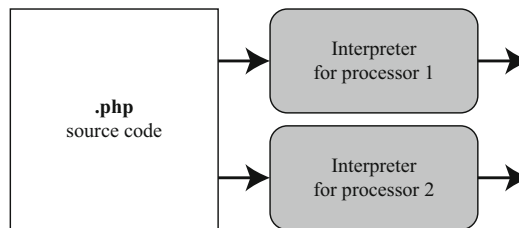


Fig. 2.3 Translating a program using an interpreter

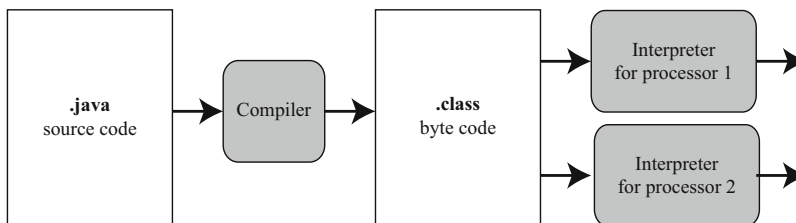


Fig. 2.4 Translating using a compiler and an interpreter

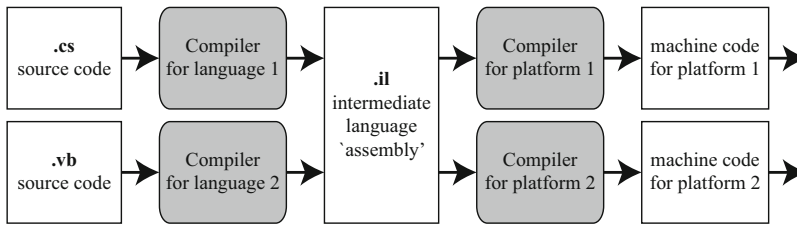


Fig. 2.5 Translating using two compilers

program just before the code is needed, this special type of compiler is also called a *Just-In-Time compiler* (or “jitter”).

2.5.6 A Final Note

In this section, we’ve shown how various languages use different translation mechanisms. The language definition itself, however, is independent of the translation mechanism. For example, when you write a program in C#, you do not necessarily have to translate it using two compilers. The Unity3D engine, for instance, uses C# as an *interpreted* script language.

2.6 Syntax and Semantics

Programming in a language such as C# can be quite difficult if you don’t know the language’s rules. The **syntax** of a programming language refers to the formal rules that define what is a valid program (in other words: a program that a compiler or interpreter can read). By contrast, the **semantics** of a program refers to its actual *meaning*.

You may recognize the word “syntax” from high-school calculators: they can respond with the message “Syntax Error” if you forget to type in a bracket somewhere. By skipping a bracket, you weren’t following the rules of the language (the *syntax*) that your calculator understands.

To illustrate the difference between syntax and semantics, take a look at the phrase “all your base are belong to us.” Syntactically, this phrase isn’t valid (a compiler for the English language would definitely complain about it). However, the *meaning* of this phrase is quite clear: you apparently lost all your bases to an alien race speaking bad English. (As you can see, the concept of syntax and semantics applies to languages in general and not just to programming languages!)

A compiler or interpreter can check the *syntax* of your program: any program that violates the rules is rejected. You could compare this to a person checking the spelling and grammar in a piece of English text: he or she can tell you whether you’ve used the language correctly. However, unlike a human reader, a compiler or interpreter can’t check the *semantics* of your program: it doesn’t know what kind of behavior you have in mind for your game or what you really want your game to *do*. So if your program compiles (meaning that it’s *syntactically* correct), you still have to test if it does exactly what you want (and only then is it *semantically* correct).

2.7 Development Cycles

We conclude this chapter by looking at the steps that a software developer typically takes when writing a program. As a programmer, you will work in **development cycles** on two different scales. The first cycle has everything to do with syntax and semantics.

2.7.1 Small Scale: Edit-Compile-Run

Programs contain many instructions. Using tools such as Visual Studio, you can type these instructions into the source files of a project, following the rules of the chosen programming language such as C#. Once you've written these instructions, you can ask the compiler to inspect our files and convert them to a program. If all is well, the compiler will create the intermediate code and then an *executable file*, which is our program in machine code.

One obvious rule is that your code should be valid C#: that is, the text you've written should follow the *syntax* rules of the programming language. If the code does not follow these rules, then the compiler produces errors and it will not create an executable file. Of course, programmers will make an effort to compile a valid (C#) program, but it's easy to make a typo, and the rules for writing correct programs are very strict. Thus, you will most certainly encounter these errors yourself as well. After a few iterations of resolving these minor errors, the compiler will have generated the intermediate code and the executable file.

When the code has successfully compiled, you can *execute* (or *run*) the program to check if your code is also *semantically* correct. In many cases, you will discover that the program doesn't exactly do what you had in mind. In a game, for example, objects may be drawn in the wrong places or respond incorrectly to the player's input. Of course, you made an effort to correctly express what you wanted the program to do, but conceptual mistakes are easily made.

When the program does not work correctly, you go back to the editor and the process starts over: you change the source code, you try to compile it until it is valid, and you run the program again to see if the problem is solved. By then, you'll probably realize that the program is indeed doing something different but still not exactly what you want. And it's back to the editor again. Welcome to your life as a programmer!

2.7.2 Large Scale: Design-Specify-Implement

As soon as your program starts becoming more complicated, it is not such a good idea anymore to just start typing away until you're done. Before you start *implementing* (i.e., writing the code and testing if it works), there are two other phases: design and specification. To explain this, let's take game development as an example.

First, you will have to *design* the game. What type of game are you building? Who is the intended audience of your game? Is it a 2D game or a 3D game? What kind of gameplay would you like to model? What kinds of characters are in the game, and what are their capabilities? Especially when you're developing a game together with other people, you're going to have to write some kind of *design document* that contains all this information, so that everybody agrees on what game they're actually developing! Even when you're developing a game on your own, it is a good idea to write down the design of the game. This phase is actually one of the most difficult tasks of game development.

Once it is clear what the game should do, the next step is to provide a global structure of the program. This is called the *specification* phase. Do you remember that the object-oriented programming

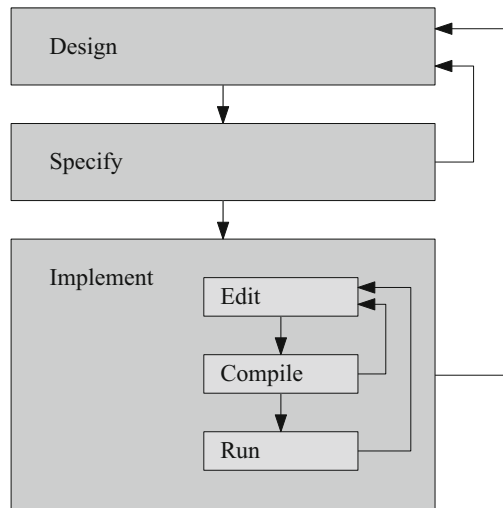


Fig. 2.6 Software development is a cyclic process on two scales

paradigm organizes instructions in methods and methods in classes? In the specification phase, an overview is made of the classes that are needed for the game and the methods that are in these classes. At this stage, you only need to describe what a method will do and not yet how that is done. However, keep in mind that you should not expect impossible things from your methods: they have to be implemented later on.

Once the specification of the game is finished, you can start the implementation phase, which means that you will probably go through the “edit-compile-run” cycle a couple of times. Once all that is finished, you can let other people play your game. In many cases you will realize that some ideas in the game design do not really work out that well. So, you start again with changing the design, followed by changing the specification and finally a new implementation. You let other people play your game again, and then, well, you get the idea.

In short, the “edit-compile-run” cycle is contained within the third step of a larger cycle: the “design-specify-implement” cycle. Figure 2.6 summarizes these two development cycles.

In traditional software engineering, the “design-specify-implement” cycle could take a long time. Programmers would spend weeks on software specifications before writing a single line of code. Nowadays, it is common to make cycles as short as possible and to integrate it directly with user testing. This is also called an *agile* software engineering approach, since it allows for more flexibility and control during the development process. By involving users early in the development stage, you learn more about what people like (or don’t like) about your game. Nowadays, game companies often release beta versions of their game to the public to gauge the players’ responses. Steam even has an early-access feature, allowing players to buy a game that’s not even finished yet!

2.8 What You Have Learned

In this chapter, you have learned:

- how computers work and that computers consist of memory (for storing things) and processors (for computing things and changing the memory);

- how programming languages have evolved from Assembler languages to modern programming languages such as C#;
- how game engines can make the life of game programmers more comfortable;
- how a computer translates your program (written in a programming language) into something that the computer understands, using a compiler or an interpreter;
- the difference between syntax and semantics;
- how the software development process typically works.

2.9 Exercises

1. *Syntax and Semantics*

What is the difference between the *syntax* and the *semantics* of a program? And how does this relate to the “edit-compile-run” development cycle?

2. *Programming Paradigms: Overview*

Some programming paradigms are subsets of other paradigms. For example, every procedural language is also imperative.

- a. Draw a diagram that shows how the programming paradigms from Sect. 2.2 are related. Include the following paradigms: *declarative*, *functional*, *imperative*, *logical*, *object-oriented*, and *procedural*.
- b. Where in this overview do famous programming languages belong? Draw the following languages at the appropriate place in your diagram: C#, Java, C, BASIC, Haskell, and Prolog.

3. *Programming Paradigms: Statements*

Indicate whether the following statements are true or not (and explain why):

- a. All imperative languages are also object-oriented.
- b. All object-oriented languages are also procedural.
- c. Procedural languages have to be compiled.
- d. Declarative languages cannot run on the same processor as imperative languages, because imperative languages have the assignment instruction and declarative languages don't.

4. * *The Compiler as a Program*

The translation of a piece of code in a programming language (such as C#) to machine code doesn't just happen magically: it's a process that *programmers* have defined and created. Thus, a compiler itself is also a program. But if a compiler is a program, then someone must have compiled it at some point:

- a. Can a compiler itself be written in a programming language? And if so, can that language be the same as the language that the compiler compiles?
- b. Can a compiler compile itself?
- c. What language do you think has been used to write the first compiler ever?

Chapter 3

Game Programming Basics



This chapter covers the basic elements of programming games. It provides a starting point for the chapters that follow. First, you will learn about the basic elements of every C# program. We'll take a very small console application as a first concrete example.

Next, we'll discuss the basic skeleton of any game: the game world and the game loop. You will then learn to identify all these elements in a simple MonoGame application.

Finally, you'll learn a bit about the organization of larger codebases, and we will explain how you can clarify your code by using comments, layout, and whitespace in the right places.

3.1 Structure of a C# Program

Let's begin by identifying the most important elements of any C# program. Overall, a program consists of *instructions*, which are grouped in *methods*, and these methods are (in turn) grouped in *classes*. We will go a bit deeper into these concepts now.

This overview may seem overwhelming, but don't worry: throughout this book, you will learn the details step by step.

3.1.1 Instructions

As explained in Chap. 2, C# is an *imperative* language. This means that it uses **instructions** to define the actual tasks that a program needs to do. Instructions are executed one after the other, and each instruction changes the memory or shows something on the screen.

In Chap. 1, you created your first MonoGame project, and you changed the line `GraphicsDevice.Clear(Color.CornflowerBlue);` to `GraphicsDevice.Clear(Color.Chocolate);`. This line was an instruction—and indeed, by changing it, you made the program do something different. So, in short: instructions are the parts of a program that do the actual work.

In C#, an instruction always ends with a semicolon (the; symbol). By typing a semicolon, you tell the compiler that an instruction ends and that a new instruction (or some other new part of the code) will begin.

3.1.2 Methods: Grouping Instructions Together

Because C# is also a *procedural* language, instructions are grouped in **methods**. Every instruction in a program belongs to a method. The group of instructions inside a method is called the **body** of the method. Above the body, you write the **header** of the method, which contains the method's name and other things that we'll talk about later. As a programmer, you can choose any name for your methods. For example, if you create a platform game and you write a list of instructions that let your game character jump, it makes sense to group these instructions in a method with the name `Jump`.

There's one important thing to keep in mind: the instructions in a method are only executed when you *call* that method. For instance, if you've created a method `Jump` that lets your character jump, this jumping will not happen unless you write the line `Jump()`; somewhere else in your code. Such a line is a **method call**, and (in this case) it will make sure that your character actually jumps. By contrast, the method `Jump` only defines what *should* happen when the method is called.

So: a method is a list of instructions that your program *can* perform. These instructions are performed as soon as you *call* the method.

3.1.3 Methods and Parameters

In Chap. 1, the instruction `GraphicsDevice.Clear(Color.CornflowerBlue);` that you changed was also a method call. More precisely, it was a call to a method named `Clear`, which is defined somewhere in the `MonoGame` engine. When you changed `Color.CornflowerBlue` to `Color.Chocolate`, you were still calling the same `Clear` method, but you gave this method different “details” to work with. In this case, the detail you changed was the exact color that you asked the method to draw.

Formally, you changed a **parameter** of the `Clear` method. The `Clear` method itself doesn't specify which color it draws. Instead, it allows you to choose your own color and give it to the `Clear` method as a parameter. This is a very nice concept because it lets you use the same `Clear` method to draw any color. In C#, when you call a method, you write the parameter values between brackets. If a method has no parameters, you simply write a pair of brackets with nothing in between.

Do we need to know which instructions are grouped together in the `Clear` method in order to use it? No, we don't! This is one of the nice things of grouping instructions in methods. You (or other programmers) can use the method without having to know how it works. By smartly grouping instructions in methods, and methods in classes, it is possible to write reusable pieces of program that can be used in many different contexts. The `Clear` method is a good example of this. It can be used for a lot of different applications and you don't need to know how the method works in order to use it. The only thing you need to know is that it takes a color as a parameter.

You'll learn much more about methods and parameters in Chap. 7. For now, just remember that a method is a group of instructions, that these instructions are executed when you call the method, and that methods often have parameters.

3.1.4 Classes: Grouping Methods Together

Finally, because C# is also an *object-oriented* language, methods are grouped in **classes**. Very roughly speaking, a class is a group of methods that describes what a certain part of your program can do. Just like how instructions must be part of a method, methods *must* be part of a class. And just like

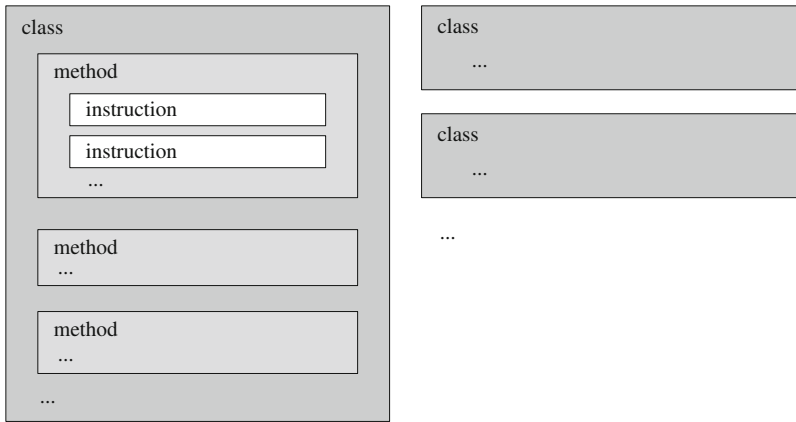


Fig. 3.1 The overall structure of a C# program, consisting of classes, methods, and instructions

methods, classes have a **body** (the actual contents of the class) and a **header** (the name of the class plus some other information). You can give classes any name that you want.

In game programming, we often use a class to describe a particular object that moves in the game. In your hypothetical platform game, it probably makes sense to create a class `Character` that describes all the things that your game character can do. The `Jump` method would then be one of the methods within this class, because jumping is part of your character's behavior.

In this book, you'll start with example programs that consist of a single class. Later (in Chap. 7), you will organize your code into multiple classes, but let's not get ahead of things. For now, just remember the overall structure as shown in Fig. 3.1: instructions are grouped into methods, methods are grouped into classes, and all these classes together form your program.

3.1.5 A Class Is a Blueprint for Objects

Remember that a method is a list of instructions that only gets executed when you *call* that method. Similarly, a class is a list of methods (plus some extra data) that doesn't do anything until you create an **object** with it. An object that you create with a class is called an **instance** of that class. Thus, it's useful to think of class as being a *blueprint* for the objects in your program.

This also means that you can reuse the same blueprint for multiple objects! For instance, if there are multiple controllable characters in your game, you use the same `Character` class to describe what all of these characters can do. The multiple characters are different *objects* of the game and different *instances* of the `Character` class. But in general, their behavior is described by a single class in code. You'll learn much more about classes and instances in Chap. 7 and onward.

3.2 Hello World: The Smallest C# Program You Will Ever See

As a concrete example, take a look at the program in Listing 3.1. This is pretty much the smallest C# program that someone can write. It is a console application that writes the text "Hello World!" to the console and then closes itself again when the user presses any key.

Listing 3.1 The HelloWorld console application

```
1 class HelloWorld
2 {
3     static void Main()
4     {
5         System.Console.WriteLine("Hello World!");
6         System.Console.Read();
7     }
8 }
```

The curly braces ({ and }) that you see in this program are very important: they define where the *body* of a class or method begins and ends. These curly braces are always nicely nested, because methods are always contained inside classes and methods are always detached pieces of code.

The line immediately before an opening brace (a { symbol) is the *header* of the class or method. In this program, line 1 (**class HelloWorld**) is a class header: it says that the program contains a class with the name HelloWorld. Line 3 (**static void Main()**) is a method header: it says that the class HelloWorld contains a method with the name Main (you can ignore the words **static void** for now).

The only lines we haven't identified yet are lines 5 and 6. Can you guess what these lines are? Indeed, they are *instructions*, the lines of the program that actually *do* something. In this case, line 5 writes the text "Hello World!" to the console, and line 6 waits for the user to press a key and then closes the program. Both of these lines are method calls. The call to `System.Console.WriteLine` has a single parameter, which indicates the exact text that you want to write. The call to `System.Console.Read` has no parameters: it always does the same job when you call it.

In short, HelloWorld is a program with a single class (called HelloWorld), which contains a single method (called Main), which contains two instructions (or more specifically method calls) that do something with the console.

The Main Method — We said before that you can give your own methods any name you want. However, every program needs exactly *one* special method that will be called when the program *starts*, to get the program up and running. The compiler needs to know the name of this method, so that it knows what the program's very first task will be.

By default, every new Visual Studio application already contains a start-up method with the following header: **static void Main()**. You can see this in the HelloWorld example as well. The project settings in Visual Studio already store the fact that Main is the start-up method. Technically, you're allowed to change the name Main into something else—but then you'll have to change this in the project settings as well. So, it's easiest if you just keep that method's name as it is.

In the rest of this book, we'll say that Main is the start-up method that every program needs. But officially, every program needs *a* start-up method, and it's called Main by default.

So, Main is the very first method that gets called when the program starts. When Main gets called, the memory for your program is still completely empty: not a single object exists yet. In MonoGame applications, Main is responsible for creating an object that will represent the actual game, and that's all it will do; the rest of the work will be done by the game itself.

In very simple *console* applications, such as our HelloWorld example, you may not need any other methods than Main. As soon as your program becomes larger, it makes sense to introduce more methods, classes, and so on—but Main will always be somewhere in your program.

The HelloWorld program is also one of the example projects of this chapter. As an exercise, open this program in Visual Studio, try to change a few things about it, and then compile and run the program (by pressing F5). Here are a few suggestions and their effects:

- If you change the word HelloWorld on line 1 to something else, such as GoodbyeEveryone, the program will still do exactly the same. The only difference is that the class now has a different name. This can affect you as a programmer (in case you ever want to use the class somewhere else in the program), but you don't see it if you run the program.
- If you change the text "Hello World!" on line 5 to something else, such as "Goodbye, everyone!", your new message will be printed instead of the original one. This is because you have changed the *parameter* of the WriteLine function.
- If you remove line 5, nothing will be printed to the console. The only remaining instruction is the one that waits for the user to press a key.
- If you remove line 6, the text "Hello World!" from line 5 will still be printed, but the program will no longer wait for a key press. As a result, you'll hardly be able to see the message because the program immediately stops again.¹
- If you remove any of the other words or brackets, there's a good chance that Visual Studio cannot compile the program anymore. You will then receive error messages because your program is no longer syntactically correct; in other words, you are no longer following the strict rules of the C# language. Don't worry—by simply programming more often, you will gradually learn what's allowed in C# and what isn't.

3.3 Building Blocks of a Game

We've said before that games are special types of computer programs. This section talks about the two main building blocks of a game program: the *game world* in which the game takes place and the *game loop* that continuously updates this game world and draws it on the screen. These building blocks are always part of a game, no matter what programming language is used.

3.3.1 The Game World

What makes games such a nice form of entertainment is that you can explore an imaginary world and do things there that you would never do in real life. This imaginary realm in which you play the game is called the **game world**. Game worlds can range from very simple domains (such as the rectangular grid in *Tetris*) to very complex virtual worlds (such as in games like *Grand Theft Auto* and *World of Warcraft*).

When a game is running, the computer maintains an internal representation of the game world. This representation doesn't look anything like what you see on the screen when you play the game. It consists mostly of numbers describing where all objects are located, how many hit points an enemy can take from the player, how many items the player has in his/her inventory, and so on. Fortunately, the program also knows how to create a visually pleasing representation of this world to display on the screen. Players never see the internal representation of the game world, but game developers do. When

¹We lied about HelloWorld being the smallest possible program. If you remove lines 5 and 6, you will still have a valid C# program that you can compile and run. But what's the fun of a program that only starts and stops without doing anything?

you want to develop a game, you also need to design how to represent your game world internally. And part of the fun of programming your own games is that you have complete control over this.

3.3.2 The Game Loop

An important thing to realize is that the game world *changes* while the game is running: monsters move to different locations, enemies get killed, objects explode and make sounds, and so on. Furthermore, the player *influences* how the game world changes, for example, by pressing buttons on a controller. Therefore, simply storing the game world in the computer's memory is not enough: the game world needs to be kept up-to-date based on everything that is going on. In addition, the updated game world should be displayed on the screen, so that the player can see what has changed.

To ensure a smooth experience for the player, this process of updating and showing the game world needs to be repeated constantly. This repeating process is called the **game loop**. The game loop is responsible for two categories of tasks: updating and maintaining the game world and displaying the game world to the player. Throughout this book, we'll call the first task the Update method, and we'll call the second task the Draw method.

The game loop continuously performs these two methods, one after the other: update, draw, update, draw, update, draw, and so on. Each call to the Update method simulates that some time has passed, and each call to the Draw method shows the player a "snapshot" of what the game world currently looks like.

As an example of how the game loop works, let's look at the game *Pac-Man*. The game world of this game is of a labyrinth with a few ghosts moving around and a number of white dots ("pills") to pick up. At any point in time, Pac-Man is located somewhere in this labyrinth and is moving in a certain direction.

In the Update method, the game should check whether the player is pressing an arrow key. If so, the position of Pac-Man should be updated according to the direction the player is requesting. This move can have many consequences. For instance, if Pac-Man now touches a white dot ("eats a pill"), this dot should disappear and the player's score should increase. If this dot is a power-up, the game should make sure that all ghosts are going to behave differently. Also, if this dot was the last remaining dot in the labyrinth, the game should recognize that the player has finished the level.

And this is just the part of Update that's related to the player's actions. Many other things should happen as well: the game should update the positions of the ghosts, it needs to check whether Pac-Man collides with any of the ghosts, and so on. You can see that even in a simple game like Pac-Man, the Update phase is already quite complex!

In the Draw method, the game should (of course) draw the game world: the labyrinth, all characters at their freshly updated positions, and all dots that have not yet been picked up. Next to this, the game should show other information that is important for the player to know, such as the score, the remaining number of lives, and so on. This extra information can be displayed in different areas of the game screen, such as at the top or the bottom. This part of the display is sometimes also called the *heads-up display* (HUD).

Modern 3D games have a much more complicated set of drawing tasks. These games need to deal with lighting and shadows, reflections, culling, visual effects like explosions, and much more. However, the main idea remains the same as in our Pac-Man example: the game loop should (in some way) keep displaying the game world in its most recent state.

A single round of the game loop is often called a **frame**. Many game engines try to run their game loop at a consistent speed, for example, 60 frames per second. This kind of game loop is called a **fixed-timestep** loop. The alternative option is that the game engine tries to execute the loop as often as

possible: this is called a **variable-timestep** loop. The MonoGame engine tries to run a fixed-timestep loop of 60 frames per second. It's possible to change this to a variable-timestep loop, but we will not go into this further.

3.3.3 How the Game Loop Gives the Illusion of Movement

As another example, imagine a simple game where a balloon is drawn at the position of the mouse pointer. When you move the mouse around, the balloon moves along with it. You could implement this as follows. In the Update method, you write an instruction that retrieves the current position of the mouse pointer and that stores it in memory. In the Draw method, you write an instruction that displays a balloon image at the stored position.

You might wonder why this would work: you're not actually moving a balloon, but you're simply drawing it at a different position in each frame. Still, for the player, it looks like something is moving smoothly. This is the power of the game loop: by showing different images at a high framerate (such as 60 times per second), it lures the player into thinking that everything moves smoothly. It's exactly how movies work as well: by showing many different images quickly after each other, you can create the illusion of movement.

3.4 Structure of a MonoGame Application

You now know a bit about instructions, methods, and classes: the main elements of a C# program. You also know that games have a game loop: a pair of methods (Update and Draw) that are being called over and over again, to give the player the illusion of smooth movement. Knowing this, let's go back to the simple MonoGame application from Chap. 1, which only drew a background color on the screen. For this chapter, we've supplied a stripped version of this default program, called BasicGame. The code of this program is shown in Listing 3.2. You can find it as an example project of this chapter as well.

Listing 3.2 A very basic game application

```
1 using Microsoft.Xna.Framework;
2 using Microsoft.Xna.Framework.Graphics;
3 using System;
4
5 class BasicGame : Game
6 {
7     GraphicsDeviceManager graphics;
8
9     [STAThread]
10    static void Main()
11    {
12        BasicGame game = new BasicGame();
13        game.Run();
14    }
15
16    public BasicGame()
17    {
18        graphics = new GraphicsDeviceManager(this);
19    }
20
21    protected override void LoadContent()
22    {
```

```
23 }
24
25 protected override void Update(GameTime gameTime)
26 {
27 }
28
29 protected override void Draw(GameTime gameTime)
30 {
31     GraphicsDevice.Clear(Color.Olive);
32 }
33 }
```

With the knowledge you have so far, you can clearly identify the methods `Update` (lines 25–27) and `Draw` (lines 29–32), which represent the game loop. Just like in the `HelloWorld` example, these methods both have a header that includes the method’s name, and they both have a body that’s nicely sealed off by a pair of curly braces.

There is also a method named `Main`. After all, `BasicGame` is still a C# program, and all C# programs require a start-up method (which is usually called `Main`).

You also see that the word `Main` is followed by two brackets with nothing in between, while the words `Update` and `Draw` are followed by the phrase `GameTime gameTime`. Comparable to the `HelloWorld` example, this means that the `Update` and `Draw` methods have a *parameter* (while the `Main` method does not). In this case, `gameTime` is the *name* of this parameter (chosen by the programmer), and `GameTime` is the **type** of this parameter. The type of a parameter indicates what kind of information the parameter contains, such as a number, a color, or (in this case) information about how much time has passed since the last frame. You’ll learn more about types in the next chapter.

Furthermore, all methods are grouped in a class called `BasicGame` (line 5). But this time, line 5 contains more than just the name of our class. The line **class** `BasicGame:Game` means that the class `BasicGame` is a “special version” of another class, called `Game`. You don’t have to understand this completely yet, but you should know that the `MonoGame` engine has already defined a `Game` class for us. By saying that `BasicGame` is a special version of `Game`, we get a lot of `MonoGame` functionality for free, including the game loop.

3.4.1 The Game Loop in `MonoGame`

When you run the `BasicGame` program, the `Update` and `Draw` methods are continuously called at a very high speed. Because a method is basically a group of instructions, every time the `Update` method is called, the instructions inside that method are executed. The same goes for the `Draw` method.

In our program, the `Update` method does not contain any instructions yet, so nothing concrete will happen when `Update` is called. The `Draw` method currently contains only one instruction that fills the game window with a particular color. It does this over and over again in each frame of the game loop, but you can’t see this as a player because the result looks the same in every frame. So although nothing interesting seems to be going on, there really is a game loop running in the background when you play the game.

Wait a second—this might seem weird. We said earlier that a method is not executed unless you call it somewhere else in your program. But even though you’re not calling `Update` and `Draw` yourself, `BasicGame` still knows when to execute these methods. How is that possible?

Well, this is one of the things that the `MonoGame` engine already does for you. In the `Game` class that `MonoGame` has provided for us, the `Update` and `Draw` methods already exist, and there’s already a

game loop built in that calls `Update` and `Draw` over and over again. By saying (on line 5) that our own class `BasicGame` is a special type of `Game`, we get all methods from the `Game` class for free. The only thing you have to do yourself is add *instructions* to these methods. So the line **class** `BasicGame`: `Game` is quite important for letting the game do what it does!

In object-oriented programming languages such as C#, this concept is called **inheritance**: letting a class “inherit” the behavior of another class, so that you can make a specialized version of it. Inheritance is one of the most powerful features of object-oriented programming, but it’s already a pretty advanced topic, so don’t worry if it’s too much to take in right now. We will not dive deeper into inheritance until Chap. 10. Until then, you should at least know that inheritance is extremely useful and that it’s the main reason that we can use the game skeleton provided by MonoGame.

3.4.2 Other Methods of the *Game* Class

The game loop is not the only thing that MonoGame gives you for free. Next to `Update` and `Draw`, MonoGame incorporates a few other useful methods that you automatically get by inheriting from the `Game` class. Two of these methods are related to starting the game, and the difference between these two methods is a bit complicated.

The `Initialize` method is executed once, when the game begins. This is the place where you can do initialization tasks, such as changing the width and height of the game window, setting up an input device, or opening a network connection. The `Initialize` method is added automatically in every new project that you create. In our `BasicGame`, we have removed it again because we don’t need it for such a simple example.

The `LoadContent` method is also executed once: *after* the `Initialize` method but *before* the game loop starts. In this method, MonoGame has set up a `ContentManager` object, which is in charge of managing game assets such as sprites, sounds, or other files that the game needs. Therefore, `LoadContent` is the first place where you can load sprites and sounds. We’ll explain how to do this in Chap. 5.

If we wanted to, we could also load our assets directly in the `Update` or `Draw` method. However, this would significantly affect the performance of our game, since we would then reload these files 60 times per second. It’s smarter to load game assets only once into the internal memory, and `LoadContent` is the first place where you can do this.

Because the `Initialize` and `LoadContent` methods are executed shortly after each other, you could also leave out `Initialize` and just use `LoadContent` for everything. This is what we have done in the `BasicGame` example, and we will keep doing it throughout this book. You’re free to keep using both methods, though. The only thing to watch out for is that some things (such as loading game assets) cannot be done in `Initialize` yet.

Finally, the `LoadContent` method also has a counterpart called `UnloadContent`, and you might already guess what its purpose is. `UnloadContent` is called after the game loop ends (which happens when the player quits the game). In this method, you can clean up any memory that was allocated for game assets.

We will not use the `UnloadContent` method in this book. In our examples, quitting the game always means closing the application, and the memory will be cleaned up automatically when the application is closed. However, `UnloadContent` can be useful in a more complicated application where players can exit the game without immediately closing the application. You could imagine a sort of “hub program” in which the player can choose between several different games. If the player quits one of these games and returns to the hub, it makes sense to clean up the assets of the game that has just been closed.

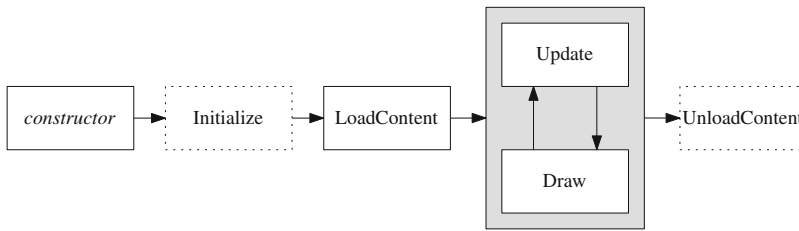


Fig. 3.2 The game loop and its associated methods in a MonoGame application. The `Initialize` and `UnloadContent` methods will be ignored in this book

In summary, these are the basic methods in every MonoGame application:

1. `Initialize` for initialization tasks such as opening a network connection or setting up an input device;
2. `LoadContent` for loading sprites, sounds, or other assets needed for the game;
3. `Update` for continuously updating the game world according to the time passed and to what the player is doing;
4. `Draw` for drawing the game world onto the screen;
5. `UnloadContent` for unloading game assets just before the game closes.

There's one more method in `BasicGame` that we haven't discussed: the method named `BasicGame` (lines 16–19). This method is not part of the MonoGame engine. Instead, it's a so-called **constructor** method that will be called as soon as the game is created by the `Main` method. Don't worry about this for now; you'll learn all about constructors in Chap. 7.

Figure 3.2 shows in what order these methods are executed. Every game that you are going to create in this book follows this basic structure (or a simplified version of it). In this book, you will see lots of ways to fill these methods with the tasks that you need to perform in your game. Along the way, you'll learn a lot of general programming techniques that are useful for games (and for other applications).

3.4.3 The Graphics Device

There's one last important thing that belongs to every MonoGame application. Before we can draw anything, we need to initialize the so-called *graphics device*. This device controls the graphic capabilities of the game, so it's very important! Initializing the graphics device is something that you need to do once, before the actual game starts. Otherwise, you cannot draw anything on the screen.

This initialization step is always done in the constructor method (which is called `BasicGame` in this case). In `BasicGame`, we initialize the graphics device on line 18:

```
graphics = new GraphicsDeviceManager(this);
```

From that point on, you can use the graphics device to (for example) clear the screen and set it to a particular background color. You can see this happening on line 31, in the `Draw` method:

```
GraphicsDevice.Clear(Color.Olive);
```

`Clear` is a method of the `GraphicsDevice` class. To call it, you need an *instance* of the `GraphicsDevice` class to work with. The `Game` class of MonoGame stores such an instance, under the name `GraphicsDevice`. Therefore, the code `GraphicsDevice.Clear` will call the `Clear` method for that instance. And as you've seen before, `Color.Olive` is a *parameter* of the `Clear` method.

3.5 Structure of a Larger Program

The HelloWorld example from Sect. 3.2 was extremely small. The BasicGame example is already a bit bigger, mostly because it uses the MonoGame engine in the background. You'll notice that programs will become more complex throughout this book (and throughout your life as a programmer!). To be prepared for this, it's good to know about a few concepts that you'll come across in larger C# programs.

3.5.1 Namespaces: Grouping Classes Together

Classes can be grouped into so-called **namespaces**. Namespaces are a way to group related classes, just like how classes are a way to group related methods. As usual, a namespace has a header (the word **namespace** followed by a name of your choice) and a body enclosed by braces.

Namespaces are useful for grouping classes that are strongly related to each other. For example, you could create a namespace called `Graphics` that contains all classes that have something to do with displaying things on the screen. Let's say you've written a class `Drawer` within this namespace `Graphics` (and you've chosen the name "drawer" because the class is responsible for "drawing things"). If you want to use this class somewhere else in your code, you'd have to write `Graphics.Drawer` to specify exactly which class you're talking about. This is necessary to avoid conflicts: if somebody has already written a `Drawer` class somewhere else in the program,² the compiler does not automatically know which version you're talking about, so you need to help it a bit. However, if you want to use your class somewhere *within the same namespace*, you can just write `Drawer` because then the compiler understands where you are.

Based on this explanation, you may think that namespaces only make things more complicated—but they're actually a very nice way to organize larger projects into groups of classes. Namespaces can also contain other namespaces, so you can create a hierarchy of groups that's (conceptually) comparable to a folder structure on your hard drive.



Quick Reference: Namespaces

To let one or more classes be part of a namespace, write the following lines around it:

```
namespace MyNamespace
{
    ...
}
```

where `MyNamespace` should be replaced by your namespace name of choice and `...` should be replaced by the code of your class(es).

If your class is called `MyClass`, you should refer to it as `MyNamespace.MyClass` whenever you're not inside the same namespace. If you *are* inside the same namespace, you can simply write `MyClass`.

Namespaces can also be grouped into each other.

²Not unthinkable if your game is somehow related to furniture.

3.5.2 Libraries: Relying on Other Classes

Take a look at lines 1–3 of `BasicGame`. These lines don't fit into any of the categories we've told you about: they are not class headers, method headers, or instructions. Instead, these lines indicate that the program may use classes or methods from three **libraries**: `Microsoft.Xna.Framework`, `Microsoft.Xna.Framework.Graphics`, and `System`. A library is a set of classes (usually created by someone else) that you can reuse in your own program.

The `Microsoft.Xna.Framework` library defines the `Game` class that our game inherits from. If we hadn't indicated that we want to use this library, the compiler wouldn't have known that a `Game` class even exists. We would then have to write `Microsoft.Xna.Framework.Game` everywhere instead of `Game`, to indicate where the class comes from. Thus, by writing **using** `Microsoft.Xna.Framework`; at the top of your file, you can make the rest of your code easier to read.

Similarly, `Microsoft.Xna.Framework.Graphics` defines the `GraphicsDevice` class that is used at a few points in the program, and `System` is a library that is often used in Windows programs. If you want, you can try to remove these lines, and you'll see that Visual Studio will complain about words that it suddenly doesn't know anymore.



Quick Reference: Libraries and using

If you want to use classes or namespaces that are defined somewhere else (such as in an external library), write the following line at the beginning of a file:

```
using LibraryName;
```

where `LibraryName` should be replaced by the name of the class or namespace you want to use. For example, the line **using** `Microsoft.Xna.Framework`; includes the `MonoGame` library, which allows you to use the `Game` class (among other things). `LibraryName` can also be the name of a namespace you've defined yourself.

If you don't specify these things via **using**, you have to write the word `LibraryName` more often in the rest of your code. This is also valid C#, but it makes your code longer and (usually) harder to read.

Libraries are very useful, because they allow you to *reuse* methods and classes written by other people. This way, you don't have to reinvent the wheel. In `BasicGame`, we use the `Clear` method on line 32 to clear the window and give it a color. This method is not written by us, but we can still use it because we indicated in the beginning of our program that we need things from the `Microsoft.Xna.Framework.Graphics` library.

Dealing with Errors — If you ever forget to include a certain library that your code needs, the compiler will show errors. For example, if you forget to include `Microsoft.Xna.Framework`, the compiler will not know what you mean by the word `Game`.

Luckily, if this happens, Visual Studio is smart (and nice) enough to help you out. It can suggest what you probably need to change in your program to resolve the error. In this same example, if you hover your cursor over the word `Game`, you will see an option called *Show*

(continued)

potential fixes. Click that option to see Visual Studio's suggestions for fixing the error. In this case, you have (at least) two options: replace `Game` by `Microsoft.Xna.Framework.Game` or include the `Microsoft.Xna.Framework` library at the top of your file.

Actually, every time your code contains an error (or a warning), you can use this trick to see how Visual Studio can help you out. The suggestions you receive don't always solve your problem, but they can at least point you in the right direction.

Don't get scared when your code contains errors! For a very large part, programming consists of (re)writing code step by step. While you're still writing, you will see many errors appear and disappear. If there are still errors when you *think* your code is finished, then you can use those remaining errors to easily see what problems you still need to solve—and most of the time, you'll even get hints on *how* you can solve them.

In other words, try to imagine compiler errors as hints for what you still need to do. That way, dealing with errors becomes less intimidating.

3.5.3 Compilation Units

C# programs are stored in text files. It's common to create a separate text file for each class, but you could also put multiple classes in a single file if you want to. A text file containing C# code is called a **compilation unit**. In the `BasicGame` example, there is only one compilation unit, and it contains a single class.

A compilation unit consists of (optionally) a number of **using** instructions, followed by a number of namespaces or classes. These namespaces and classes are sometimes also called *top-level* elements, because they don't have to be contained inside a pair of curly braces (i.e., inside the body of another piece of code).

3.6 Program Layout

This section deals with the layout of a program's source code. You'll first see how to add clarifying comments to your code. Then you'll learn how to write instructions as clearly as possible by using single or multiple lines, whitespace, and indentation.

3.6.1 Comments: Explaining What You're Doing

For human readers, it can be quite confusing to understand a program that only consists of a pile of code. Luckily, most programming languages (including C#) allow you to write **comments** that can help clarify what your program does. This is especially useful if another person has to understand the code you've written or if you look back at your own code next month and don't remember all the details. Comments are completely ignored by the compiler: they're simply meant to the program more understandable for humans.



Quick Reference: Comments

There are two main ways in C# to mark comments in your code:

- If you write `//` somewhere on a line, everything from that point until the end of the line will be ignored.
- If you write `/*` and `*/` anywhere in the file, everything between these symbols will be ignored. This allows you to write multiple lines of comments in a single block.

It's useful to place comments in your code as much as possible, to explain the purpose and meaning of your instructions, methods, and classes. Keep in mind that comments are meant to *clarify* the code: you can assume that readers of your code are familiar with C#, but you can still help them understand why you chose to write your code in a particular way. To illustrate this, compare the following piece of code:

```
// Set the background color to olive green.  
GraphicsDevice.Clear(Color.Olive);
```

to this piece of code:

```
// Pass the value Color.Olive to the Clear method of the GraphicsDevice object.  
GraphicsDevice.Clear(Color.Olive);
```

Clearly, the first comment is helpful because it explains our intentions. The second comment probably won't give programmers any information that they couldn't already figure out themselves.

Comments are also an easy way to *temporarily remove* instructions from the program. If you want to disable a piece of code for testing purposes, you can “comment it out” (e.g., by putting it `/*` and `*/` around it). If you then compile the program again, the compiler will act as if that piece of code doesn't exist. The advantage of using comments here is that you don't have to permanently remove the text from your file. If you ever want to put the code back again, you can simply remove the `/*` and `*/` symbols.

However, if you finish your program or share the code with other programmers, it's good practice to remove the parts of your code that you've commented out. Otherwise, others may be confused about why the unused code is still there.

By the way: in Visual Studio, if you start a comment with *three* slashes instead of two, just above a method or class (or various other elements you'll see later), the editor will treat your comment as a special “summary” of the element below it. This is a convenient way to add **documentation** to your program: a full description of what your methods/classes do, meant for programmers to look at. You'll see more of this in Chap. 11, when we finish the first game of the book (Painter).

3.6.2 Instructions vs. Lines

There are no strict rules about how to distribute the text of a C# program over the lines in a text file. Usually, you write every instruction on a separate line. This isn't necessary for the compiler, though. For example, we could squeeze the HelloWorld program onto two lines³ like this:

```
class HelloWorld { static void Main()
{ System.Console.WriteLine("Hello World!"); System.Console.Read(); }}
```

This is still the same HelloWorld program according to the compiler. However, it's now harder to read for us humans, because (for example) you can't clearly see when the Main method starts and ends. So it's allowed to write multiple instructions on a single line—sometimes this can make the program clearer, but in many other cases the result isn't very programmer-friendly.

The opposite is also possible. If a single instruction is very long (e.g., because it contains many parameters), it sometimes makes sense to distribute the instruction over multiple lines. You will see this happening later on in this book as well.

In short, adding and removing newlines is OK according to the C# language. As long as you don't break up a word, that is—but that's more related to whitespace in general.

3.6.3 Whitespace and Indentation

As you can see, the BasicGame example uses whitespace liberally. There is an empty line between each method, as well as spaces between an equals sign and the expressions on either side of it. Just like the use of newlines, spacing can help to clarify the code for the programmer, but it has no meaning for the compiler.

The only place where a space is really important is between separate words: you aren't allowed to write "**protected override void** Update" as "protectedoverridevoidUpdate". And similarly, you aren't allowed to write an extra space in the middle of a word.

In text that is interpreted literally, spaces are also taken literally. For instance, there's a difference between writing

```
Content.RootDirectory = "Content";
```

and

```
Content.RootDirectory = "C o n t e n t";
```

But apart from this, extra spaces are allowed everywhere. The following are good places to put extra whitespace:

- Behind every comma and semicolon (but not before).
- Left and right of the equals sign (=), such as in the `Content.RootDirectory = "Content";` instruction.
- At the beginning of lines, so the bodies of methods and classes are *indented* with respect to the braces enclosing the body.

Visual Studio often helps you a bit by automatically performing the indentation. Also, the editor automatically places whitespace at certain spots in your code to improve readability.

³We could even put the program on a single line, but that would break the layout of this book!

3.7 What You Have Learned

In this chapter, you have learned:

- that a C# program consists of instructions, which are grouped in methods, which are grouped in classes;
- that a game program consists of the game world and the game loop;
- how the game loop calls the Update and Draw methods to continuously update the game and show it to the user;
- how to structure a MonoGame program by using the Game class, which automatically gives you Update, Draw, and other useful methods;
- how namespaces and libraries are used in larger programs;
- the basic layout rules of a C# program, including how to place comments in your code and where to put extra whitespace to improve readability.

3.8 Exercises

1. *Instructions, Methods, Classes*

What is an *instruction*, a *method*, and a *class*? And how are these concepts related to each other?

2. *Comments*

What are the two ways to write comments in a C# program? And what are the most important reasons to use comments?

3. *The Game Loop*

Below are some questions about the game loop, the basis of every game program.

- a. What two actions are the main elements of the game loop? What is the use of these actions?
- b. What methods does MonoGame add to this? What is the use of these methods?
- c. What is the difference between a fixed-timestep game loop and a variable-timestep game loop? Can you think of an advantage and a disadvantage of both types?
- d. The Update and Draw methods are always executed in sequence. Technically, we could move all the code from the Update method into the Draw method, and leave out the Update method altogether. Why is it still useful to have different methods?

4. * *The Smallest Program (For Real, This Time)*

You've seen HelloWorld as an example of a very small C# program, and you probably have a picture of what you can remove from that program while still following the C# syntax.

If you take this to the extreme, what is the shortest C# program that you can possibly write? *Hint:* the shortest program *we* could come up with is a console application of 26 characters long (that does absolutely nothing).

Chapter 4

Creating a Game World



This chapter shows you how to create a game world by storing information in memory. It introduces *variables*, *types*, and *expressions*, and it shows how you can use them in C# to store or change information. At the end of the chapter, you'll use this knowledge to create a game world that changes its color over time.

We've already discussed memory a couple of times. You have seen how to execute a simple instruction like `GraphicsDevice.Clear(Color.Olive);` to clear the window and set its background color. In this chapter, you'll create an improved version of `BasicGame`, called `DiscoWorld`, in which the background color changes over time. You will do this by computing a new color in the `Update` method (in each frame of the game loop) and storing it in a so-called *member variable*. Afterwards, the `Draw` method will use this member variable to draw the actual color. This is how all games in this book will work: `Update` changes the (computer's representation of the) game world, and `Draw` shows the result.

4.1 Variables and Types

More precisely, we're going to store the background color in a so-called **variable**: a place in the computer's memory that stores a particular value. Programs in C# are full of variables, and the core business of game programming is to set these variables to the correct values (in the `Update` method) so that the `Draw` method can show the game world correctly.

4.1.1 Types

Each variable has a **type** (also called a **data type**), indicating what kind of value it represents, such as a number or a color. In the `BasicGame` example from the previous chapter, you've seen that methods often need extra information in the form of *parameters*. For instance, the `GraphicsDevice.Clear` method asks for a parameter that indicates the exact color to draw, the `Update` method in the `BasicGame` class wants game time information, and the `Main` method doesn't need any information at all.

Methods only work with certain types of information. For example, in the `Draw` method of `BasicGame`, it wouldn't make sense to write `GraphicsDevice.Clear(gameTime)`. The `GraphicsDevice.Clear` method expects a color to draw and not information about the game time!

We use data types to tell the compiler what type of information is expected in the program. This way, the compiler can automatically give you errors or warnings whenever you're using the wrong type somewhere. Try replacing `Color.Olive` by `gameTime` yourself, and you'll see that the compiler indeed tells you that something's wrong.

There are many types in the C# language. Some of these types represent *numbers*, such as **int** (used for whole numbers) and **double** (used for numbers with a fractional part). There are also types that don't represent numbers, but something entirely different. An example is **string**, a data type used for pieces of text.

Next to the types that *always* exist in C#, the MonoGame engine adds more types that are especially useful for games. Two examples that you've seen before are `Color` (used for colors) and `GameTime` (used for information about the time that has passed in your game).

It doesn't end there: you can even *create your own types*! But you'll learn more about that later. For now, we'll use the **int** data type as a running example because it's easy to understand. But almost everything you'll see in this chapter works exactly the same for all other types. We'll talk more about these other types in Sect. 4.3.

4.1.2 Declaring a Variable

Next to a type, every variable also has a *name*. This is the name by which the variable can be recognized, so that you can use it throughout your program to store and modify information. As a programmer, you can choose the name of a variable yourself.

A **variable declaration** is an instruction in your code that introduces a new variable with a name and type. By writing a declaration, you "promise" the program that this variable exists and that it can be used from now on. Here is an example of a declaration in C#:

```
int red;
```

In this example, `red` is the *name* of the variable that we've chosen ourselves, and **int** is the *type* of the variable. The word "int" is an abbreviation of *integer*: you use this data type to store an integer number (a "whole" number such as 1 or 42 or 1337).

After this declaration, the computer has reserved a place in memory that can store an **int**, and you refer to this place by the name `red`. Here's a simplified representation of what the memory looks like after you've declared the `red` variable:



Quick Reference: Variable Declaration

To declare a variable of type **int** with the name `myVariable`, write the following line:

```
int myVariable;
```

This reserves a place in memory (referred to as `myVariable`) that can store integer numbers. You can choose the variable name yourself. There are also many other data types than **int** that you can use.

4.1.3 Assigning a Value to a Variable

Now that you’ve told the compiler that some place in memory called `red` can be used for storing a number, you can start using that variable in your program to *actually* store a number. If you give a variable a value, this is called an **assignment** instruction. Here’s an example of an instruction that assigns the value 3 to the variable `red`:

```
red = 3;
```

After this assignment, the memory looks like this:



As you can see, the place in memory has now been filled with the value 3. When you assign a value to a variable for the first time, this is called an **initialization** of the variable.

An assignment instruction always consists of the following parts:

- the name of the variable that should be assigned a value;
- the equals sign (=);
- the new value of the variable;
- a semicolon.

So now, you’ve seen one instruction for declaring a variable and another for storing a value. But if you already know which value you want to store in a variable when you declare it, you can combine the variable declaration and assignment into a single line, like this:

```
int red = 3;
```

Whenever you assign a new value to a variable, the old value will no longer be stored there. So if we take the previous line of code and add the following line after it:

```
red = 5;
```

the variable `red` will now store the value 5 (instead of 3).

‘=’ is Not “Equals” — You can recognize an assignment instruction by the equals sign (the = symbol) in the middle. However, in C#, it’s better to think of this sign as “becomes” rather than “equals.” After all, the variable isn’t yet equal to the value to the right of the equals sign: it *becomes* that value after the instruction is executed.

If you’re new to programming, it can be pretty confusing that = doesn’t have the same meaning as in mathematics. Also, if = is already reserved for assigning values, what symbol should we use to ask the program whether or not two values are equal? Well, in Chap. 6, you’ll see that we use == for this. Don’t worry about this for now, though.

4.1.4 Declarations and Assignments: More Complex Examples

Here are a few more examples of declarations and assignments of integer variables:

```
int age = 16;
int numberOfBananas;
numberOfBananas = 2;
int a, b;
a = 4;
int c = 4, d = 15, e = -3;
c = d;
numberOfBananas = 5 + 6;
numberOfBananas = age + 12;
```

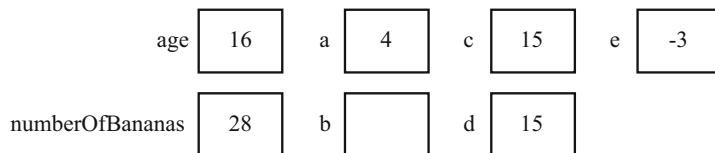
The first three lines of this example are similar to what we've seen before. In the fourth line, you see that it's possible to declare multiple variables in one declaration: this line declares two **int** variables, named *a* and *b*. In the sixth line, you see that you can even perform multiple declarations *and assignments* on a single line. (Officially, we'll call this a single declaration instruction, even though multiple variables are actually being created.)

But the right side of an assignment doesn't have to be a fixed number like 16 or 2. It can also be the name of *another variable*, such as in the line `c = d;`. This line takes the value that's currently stored in the variable *d* and then stores it in the variable *c* as well. Afterwards, the variables *c* and *d* both store the value 15. You could say that we've copied the number from one variable to the other.

The right side of an assignment can even be a more complicated *mathematical expression*, such as the sum of two numbers. In the line `numberOfBananas = 5 + 6;`, the program first computes the sum of 5 and 6 (which is 11) and then stores this number 11 in the `numberOfBananas` variable. (Remember that this overwrites the previous value of `numberOfBananas`.)

Finally, expressions can contain variables as well. In the last line, the program first computes the sum of 12 and the number that's currently stored in the `age` variable (in this case 16). This sum ($12 + 16 = 28$) is then stored in the `numberOfBananas` variable.¹

After these instructions have been executed, the memory looks something like this:



There are seven different variables, all of type **int**, and they all store particular values—except for the variable *b*, to which we haven't assigned any value yet.

¹That's a lot of bananas!



Quick Reference: Variable Assignment

Given a variable of type **int** with the name `myVariable`, the following line:

```
myVariable = 5;
```

assigns the value 5 to `myVariable` and overwrites any previous value that `myVariable` may have had. You can also assign a (first) value immediately when you declare the variable:

```
int myVariable = 5;
```

You can replace 5 by any expression of type **int**. There are also many other data types than **int** that you can use.

4.1.5 Constants

Sometimes, it's useful to explicitly state that a variable will not be changed after it has been assigned a value. Games contain many of these constant values, such as the initial velocity of a ball shot from a cannon, physical constants such as gravity, or mathematical constants such as π . Consider the following line of code:

```
const int speedLimit = 100;
```

The **const** keyword indicates that the `speedLimit` variable is a **constant**: a value that is defined once and never changes. If you try to change the variable after its initial assignment, the compiler will generate an error. In other words, if you try to execute the following instruction:

```
speedLimit = 5000;
```

the compiler will complain that you're not allowed to assign a new value to `speedLimit` because it's a constant.

4.2 Expressions: Pieces of Code with a Value

In an assignment instruction, the code fragment on the right side of the `=` sign is called an **expression**. An expression is a program fragment that has a value. It can be a simple number such as 16, a sum such as `age + 12`, a color such as `Color.Olive` (in a MonoGame program), or something more complicated.

An expression always represents a value *of a certain type*. In the previous example, all expressions (such as 16 and `age + 12`) have the data type **int**. And because they have this type, you're allowed to store their values in a variable of type **int** as well. The compiler knows this, and it protects you from using the wrong type in the wrong places. For example, the instruction `int x = Color.Olive;` is not valid, because you can't put a color in a variable that's meant to store an integer. More officially speaking, the type of the variable is **int**, while the type of the expression on the right is `Color`, and these types do not match.

There's something special going on with classes and types. The expression `Color.Olive` gives us a standard color defined by the `Color` class of MonoGame. But the *type* of this expression is `Color` as

well. So, although it may seem confusing, a class is actually also a type. How this works exactly is something that we'll discuss later on.

4.2.1 Expressions Versus Instructions

So what is the difference between an *expression* and an *instruction*? Well, an *instruction* is a piece of code that changes the memory in some way. Examples of instructions are method calls (such as `GraphicsDevice.Clear(Color.Olive);`) and variable declarations and assignments (such as `int age = 16;`). By contrast, an *expression* is a piece of code that has a value, such as `Color.Olive` and `16`. As you can see, expressions are often *used* in instructions, but expressions and instructions are not the same thing. In C#, an instruction always ends with a semicolon, so instructions are easy to recognize.

4.2.2 Operators: Combining Expressions into New Expressions

The expression `age + 12` is an example of an expression that consists of multiple parts. The parts `age` and `12` are also expressions themselves, both of type `int`. The `+` sign is a so-called **operator**: a symbol that takes expressions as “input” and computes a new expression out of it. In this case, the `+` operator takes the expressions `age` and `12` and then computes their sum. The expressions `age` and `12` are also called the **arguments** of the `+` operator.

The result of a `+` operation has type `int` again. This means that you can also store the result in a *variable* of type `int`.

An operator that calculates something based on numbers is called an **arithmetic operator**. In *int* expressions, you can use the following arithmetic operators:

- `+` for adding two numbers (e.g., `1 + 2` has the value 3);
- `-` for subtracting one number from another (e.g., `5 - 3` has the value 2);
- `*` for multiplying two numbers (e.g., `2 * 3` has the value 6);
- `/` for dividing one number by another (e.g., `12 / 4` has the value 3);
- `%` for computing the division remainder, pronounced *modulo* or *modulus* (we'll explain this operator soon).

Addition and subtraction work exactly as you'd expect. For *multiplication*, C# uses an asterisk (the `*` symbol) because the symbols normally used in mathematics (`·` or `×`) aren't found on a computer keyboard. Completely omitting the `*` operator is not allowed: for instance, you're not allowed to shorten the expression `age * 3` to `age3`. This is because numbers can also be part of variable names, so `age3` could be the name of another variable. The compiler cannot automatically know what you mean then. This is another difference to mathematics, where it's common to write $3x$ instead of $3 \cdot x$, for instance.

For *division* between integers, you should know that the result of the division will be converted to an `int` again. The result will not be rounded to the nearest whole number, but it will simply be *truncated*: the fractional part will be removed. For example, even though $14/3 = 4.666\dots$ in mathematics, the expression `14/3` in C# gives a value of 4. Similarly, the expression `3/4` gives a result of 0. So the instruction `int y = 3/4;` will store a value of 0 in the variable `y`.

The special operator `%` gives the **division remainder**. Maybe you haven't heard of this concept yet. For instance, the result of `14%3` is 2, because 12 can be divided by 3 to yield a whole number, but

14 cannot: it leaves us with a remainder of 2. Similarly, the result of $456\%10$ is 6, because 450 is a multiple of 10 and we are left with the remaining number 6.

The result of the `%` operator always lies between 0 and the value to the right of the operator. The result is 0 if there is no remainder (i.e., if the result of the division is an integer.)

Of course, the arithmetic expressions in your games will generally also contain variables (such as `age` and `numberOfBananas`) and not just hard numbers such as 16. Otherwise, you could have just calculated the results yourself, instead of writing the full expression. In this section, we simply use hard numbers to make the operators easier to understand.

The expressions on either side of an operator can be as long and complicated as you want: they can also contain other operators! But as soon as you start combining multiple operators, it's good to know something about their *priority*, to avoid surprises.

4.2.3 Priority of Operators

When you use multiple operators in an expression, the regular arithmetic rules of precedence apply: multiplication and division have priority over addition and subtraction. For example, the result of the expression $1+2*3$ is 7, and not 9. Behind the scenes, the program will *first* evaluate the expression $2*3$ (leading to a value of 6) and *then* evaluate the expression $1+6$ (leading to a final result of 7).

Addition and subtraction have the same priority, and multiplication and division as well. If an expression contains multiple operators of the same priority, then the expression is computed from left to right. So, the result of the expression $10-5-2$ is 3, and not 7: the program will first evaluate $10-5$ and then subtract 2 from that result.

When you want to deviate from this standard order, you can use parentheses (brackets) to indicate that some part of the expression has to be calculated first. For example, the expression $(1+2)*3$ gives a result of 9, because it first computes $1+2$ and then multiplies that result by 3. Similarly, $10-(5-2)$ gives a result of 7. Using more parentheses than needed isn't forbidden: for example, $1+(2*3)$ also gives a result of 7, which is exactly the same as $1+2*3$. You can go completely crazy with the parentheses if you want: for example, the expression $((1)+(((2)*3)))$ is also valid, and its result is seven again. However, an expression with way too many brackets isn't very nice to read.

This may sound complicated, but it's actually very similar to what you're allowed to type on a simple high-school calculator.

To summarize, here's a Quick Reference box:



Quick Reference: Expressions and Operators (1)

An *expression* is a piece of code that has a value of a certain type (such as `int`). An expression can be one of the following things:

- a constant value (such as 12);
- a variable;
- an operator (such as `+` or `-`) with two expressions around it;
- an expression between parentheses (to overrule the standard order of operators).

But again, note that there are many more data types than **int**, including types that don't even represent numbers. These operators don't make sense for *all* data types: for example, you can't multiply a piece of text by another piece of text.

Expressions and Method Calls — There's actually a fifth type of expression: a *method call*. So far, we've said that a method call is an instruction, such as `GraphicsDevice.Clear(Color.CornflowerBlue);`. But technically, if you remove the semicolon from that instruction, you get an *expression*. The term "method call" is often used for both the instruction (*with* the semicolon) and the expression (*without* the semicolon).

It's a bit too confusing to go into the details now. We will get back to this topic in Chap. 7, when you'll create your own methods. For now, just remember the short version of the story: when somebody says "method call," they can mean an expression *or* an instruction, depending on whether there's a semicolon behind it.

4.2.4 Variants of Operators

Some types of calculations occur very often in programs. For example, you will often want to add a certain number to a variable. Let's pretend it's your birthday. If your program contains an integer variable `age` representing your age, you'll want to increase the value of that variable by 1. So far, we've seen one way to write this down:

```
age = age + 1;
```

This instruction first computes the value of the expression `age + 1` and then stores this value in the existing `age` variable. The result is that your age will be overwritten by a number that is one larger than the previous number.²

Because it occurs so often that programmers want to add something to an existing number, there is a shorter way to write the same instruction:

```
age += 1;
```

Similarly, there are operators `--`, `*=`, `/=`, and even `%=`, which all do exactly what you'd expect.

But we can shorten our birthday instruction even more. Because you want to add a value of 1 to a variable (which is also called *incrementing*), you can also write it as follows:

```
age++;
```

In summary, `age++` is a very short way of writing `age = age + 1`. For subtracting a value of 1 (also called *decrementing*), there is a `--` operator which works in the same way. You'll see these operators being used more often in later chapters. For now, let's move to a different topic, before we accidentally make ourselves way too old.

²Happy birthday!

Kinds of Operators — In technical terms, the first operators you’ve seen in this section are all so-called **binary infix operators**. Here, “binary” means that the operators use *two* arguments, and “infix” means that the operators are always written *in between* their two arguments. Note: the word “binary” here is unrelated to the concept of “binary numbers” that you may have heard of.

The ++ and -- operators are called **unary operators** because they only have a single argument. Unary operators cannot be infix operators: after all, they cannot have two arguments around them! Instead, a unary operator is called a **prefix operator** if you write it *before* an expression or a **postfix operator** if you write it *after* an expression.

4.3 Other Data Types

So far, we’ve only looked at integers, but there are many other data types in C#. This section tells you more about these other types. First, let’s focus on the data types that represent numbers, the so-called **numeric types**.

4.3.1 The Double Type

Another numeric type that is used a lot is the **double** type. This type can be used to represent *real* numbers with fractional digits (as opposed to only whole numbers). Variable declarations and assignments work exactly the same as with **int**, except that you can use **double** to store different kinds of numbers as well. For instance, the following code:

```
double d;  
d = 3.141592653;
```

first reserves a place in memory (referred to as *d*) for storing real numbers and then stores the real number 3.141592653 at that place. (We could also have merged this declaration and assignment into a single instruction again.)

Variables of type **double** can also contain integer numbers:

```
d = 10;
```

Behind the scenes, the computer will then automatically place a zero behind the decimal point.

Dividing **double** expressions results in only small rounding errors, in comparison with **int** expressions. For instance, if the value of *d* is 10.0 and we add the following line:

```
d = d / 3;
```

the variable *d* will be updated to contain the number 3.33333333.

4.3.2 Data Types for Integers

In total, there are eleven numeric types in C#. Eight of them are especially meant for integer numbers. The difference between these eight types is the *range* of values that can be represented by each type: for instance, the type **long** can represent much larger numbers than the type **int**.

The range of an integer type is determined by the number of *bytes* that the computer reserves for a variable of that type. In the end, everything on a computer is represented by bits and bytes. A **bit** is a unit of memory that can have one of two values: 0 or 1. A **byte** is a sequence of 8 bits, which can have $2^8 = 256$ different values. So, a variable with a size of 1 byte can represent 256 possible integers.

The table below shows the eight types in C# that can represent integer numbers. For each type, we show three properties: the number of bytes reserved in memory, the smallest possible value for a variable of that type, and the largest possible value.

Type	Space	Smallest value	Largest value
sbyte	1 byte	−128	127
short	2 bytes	−32,768	32,767
int	4 bytes	−2,147,483,648	2,147,483,647
long	8 bytes	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
byte	1 byte	0	255
ushort	2 bytes	0	65,535
uint	4 bytes	0	4,294,967,295
ulong	8 bytes	0	18,446,744,073,709,551,615

The **long** type is only needed if you're planning to use extremely large values. The types **byte** and **short** are useful for representing numbers within a small range. Also, the types **short**, **int**, and **long** each have an unsigned version, of which the name begins with a *u*. Unsigned types can represent nonnegative values only (meaning 0 or higher). The **byte** type is unsigned by definition, and it has a *signed* counterpart called **sbyte**.

In general, types that can represent larger numbers also require more memory. As a programmer, it's good to think about the memory usage of your program—especially when you're developing a game for a console or mobile device with limited memory. So when you have to store a number, consider which type is best suited.

For example, let's say you have a variable that keeps track of the player's score in your game. If you know that the score is always a whole number, it makes no sense to store it as a **double**. And if you know beforehand that it's impossible for players to score billions of points, you probably don't need the **long** type either. Finally, if negative scores are impossible, it sounds like the **uint** data type will do the job.

On the other hand, your game's memory usage won't change dramatically if you change a single variable from **long** to **int**, for instance. You probably won't notice any big differences until a lot of these variables (thousands or even millions) are involved.

4.3.3 Data Types for Real Numbers

Next to the integer data types, there are three different non-integer types. These are also called **floating-point** data types. They do not only differ in the smallest and largest value that can be stored but also in their *precision* after the decimal point.

Type	Space	Significant digits	Largest and smallest value
float	4 bytes	7	+/- 3.4×10^{38}
double	8 bytes	15	+/- 1.7×10^{308}
decimal	16 bytes	28	+/- 7.9×10^{28}

Here, the **float** type uses the least amount of memory. The **double** type can store very large numbers and with high precision. The **decimal** type is even more precise, but it cannot contain values as high as the **float** or **double** types.

Every type has its own target application. The **decimal** type is useful for financial calculations or in games that require very precise physical calculations. The **double** type is used for common mathematical calculations. **float** is used if precision is not that important, but speed and memory usage is. In this book, you'll notice that MonoGame uses the **float** data type quite a lot. This is good enough for most types of games.

Floating-Point Numbers and Precision — An expression like $10/3$ should ideally give us a number of $3.33333333 \dots$, but we'd need an infinite number of decimal digits to store this value precisely. In general, because the **float**, **double**, and **decimal** types use a fixed number of bytes to represent numbers, they cannot represent all possible numbers. So, even though floating-point numbers might *look* very precise, they're actually always rounded to the nearest number that their type can represent.

If you perform lots of multiplications and divisions between floating-point numbers, then these rounding errors can build up. After many of these calculations, it's very unlikely that the outcome will *exactly* have a particular value. Depending on what happens in your program, it might be dangerous to check if a **float** variable is exactly 10.0 (for example). You probably won't run into such problems very quickly, but it's still good to be aware of this.

4.3.4 Mixing Different Data Types

When you write a real number in C#, you can explicitly indicate that it is of type **float** by adding the letter **f** behind it. For example, the following line declares a **float** variable and fills it with a **float** value:

```
float speed = 0.3f;
```

The **f** written behind the value 0.3 indicates that the value is of type **float** and not of type **double**. Similarly, you can use the letter **m** for **decimal** constants.

Sometimes, these suffixes are obligatory. Consider the following instruction:

```
float speed = 0.3;
```

This instruction would actually give a compiler error! Why? Because the compiler interprets the value 0.3 as being of type **double**. So in this line, we're (accidentally) trying to squeeze a **double** number into a **float** variable, which doesn't fit because **float** uses only half of the memory that **double** uses.

You may think that this is a pretty dumb mistake made by the compiler. After all, the compiler should be able to understand that the **double** 0.3 value should be converted into a **float** value, and just do it automatically, right? Well, if you think about it, such an automatic conversion could lead to very

dangerous situations. Suppose that you're a very rich person and your bank uses C# to store how much money you have on your bank account:

```
double account = 12345678912345;
```

Now you want to transfer this money to another account. However, a not-so-smart programmer accidentally uses the **float** type for that:

```
float newaccount = account;
```

Now, because of the reduced precision of the **float** type, the value that is actually stored is 12345678900000. In other words, you just lost 12,345 dollars!³ So, assigning a value to a variable of a different type may result in the loss of information. Whenever this happens, the compiler generates an error or warning so that you are aware of it. If you still would like to perform the assignment anyway, you have to state explicitly that you want to convert the value into a value of type **float**:

```
float newaccount = (float)account;
```

When you explicitly convert an expression of a certain type into an expression of another type, this is called a **cast**. In C#, a cast is performed by writing the type that you want to cast to between parentheses in front of the expression.

4.3.5 Data Types in General

Let's go back to something we've said before: there are many types built into C# itself, the MonoGame engine adds useful types such as **Color** and **GameTime**, and you can even create types of your own. Expressions can have any of these types. Variable declarations and assignments work exactly the same for each type.

There are two differences worth knowing about. First of all, *operators* are not defined for every possible type. It makes sense to add and multiply things that are *numeric*, and (with a little imagination) you could define the sum of two *colors*, but there are also many combinations that *don't* work. For example, the ***** operator is not defined for the **string** type. If you remember that a **string** represents a piece of text, this makes sense: it's unclear what you would mean by "multiply one piece of text by another piece of text." (By the way, the **string** type *does* support the **+** operator, but then **+** takes the role of pasting two pieces of text together.)

Second, *casting* is only possible between related types. For example, you can't cast a **Color** variable to a numeric type such as **float**, because it's not clear what such a conversion should do.

4.3.6 Classes, Instances, and Data Types

But what about the types that you can create yourself? It sounds pretty cool, but how do you *do* something like that? Well, it's a bit early to go into the details, but here's a sneak preview: every *class* that you write will automatically lead to its own type.

³That's pretty sad for someone who's celebrated multiple birthdays just a few pages ago.

Remember from Chap. 3 that a class is a blueprint for the *objects* in your program. As an example, let's say you've created a class `Character` in your game. This class does not represent a specific character yet; it just describes how characters in your game can behave. To actually *create* a character object in the game, you'll have to write something like this:

```
Character myCharacter = new Character();
```

This creates an **instance** of the `Character` class: an actual character object with which you can do things. The keyword **new** indicates that you're creating a new instance of a `Character` here. You'll see this keyword again in a few pages, when you're going to create your own color objects.

But at the same time, this line of code is just another variable declaration and assignment, just like `int red = 3;`. So, the computer reserves a place in memory (a variable) that stores the character you've just created, and `myCharacter` is just a name that we've chosen ourselves. But this time, the *type* of the variable is not **int** but `Character`!

In short, you can use the class `Character` to create *instances* of that class, and these objects will have `Character` as their type.

In fact, the extra types provided by `MonoGame` (such as `Color` and `GameTime`) are there because the programmers of `MonoGame` have created *classes* with these names. In that sense, a `Color` in `MonoGame` is no different than a `Character` you can make yourself!⁴

A data type that corresponds to a class is also called a **class type**. By contrast, the types that are not related to a class are called **primitive types**. Primitive types are always built into the C# language itself. The numeric types discussed in this chapter are all primitive types. Class types are usually created by yourself or by the programmers of a library such as `MonoGame`.

If you feel confused by all this, don't worry: you won't have to create your own classes until Chap. 7. For now, let's use our new knowledge of variables and types in a real `MonoGame` example.

4.4 Putting It to Practice: A Changing Background Color

The `DiscoWorld` example that you're going to write is basically an extension of `BasicGame` (from Chap. 3). But instead of always drawing the same background color in every frame, we're now going to draw a different color each time, based on how much time has passed in the game.

The main idea is this. In the `Update` method, we'll compute a new color that somehow depends on the game time that has passed. We'll store this color in a *variable* that can be used throughout the entire class of our game. If we use this variable in the `Draw` method, then `Draw` will always show the most recent color that we've computed. As a result, when we run the game, the color will change over time.

Let's go! To start off, create a new default `MonoGame` project in Visual Studio (as described in Chap. 1). This will generate two files with code: `Game1.cs` and `Program.cs`. You could see this as `MonoGame`'s longer version of our `BasicGame` example that you saw before. You can ignore the `Program.cs` file and focus only on `Game1.cs`.

If you run the game by pressing F5, you'll see a static background color. Let's do something about that.

⁴Technically, `Color` is a *struct* instead of a class. But a struct and a class are almost the same thing, so don't worry about that for now.

4.4.1 Using a Variable to Store a Color

Recall that `Color` is a struct that's part of the MonoGame engine. (A struct is almost the same thing as a class.) Therefore, `Color` is also a data type. So far, you've seen expressions like `Color.Olive` and `Color.CornflowerBlue`: these expressions give you specific colors, and they have `Color` as their type.

Because `Color` is a type, you can also create a variable of that type! Go to the `Draw` method, to the line that says `GraphicsDevice.Clear(Color.CornflowerBlue);`. Immediately before that line (but still after the `{` symbol), add the following line of code:

```
Color background;
```

This is a *declaration* of a variable with the type `Color`. It reserves a place in the computer's memory that can be used to store a color. (Again, `background` is just a name that we chose ourselves; other names would work as well.)

Now, let's give this variable an actual value. Change the line into this:

```
Color background = Color.Red;
```

This stores the value `Color.Red` in the `background` variable. Just like `Color.Olive` and such, `Color.Red` is a predefined color in MonoGame. It represents—you guessed it—the color red.

You can now use this variable in the `GraphicsDevice.Clear` method, because that method expects a color as a parameter. More precisely, the parameter should be an *expression of type* `Color`. The method currently takes `Color.CornflowerBlue` as its parameter, but you can safely replace that by your variable `background`, because `background` is *also* an expression of type `Color`. In other words, replace this line:

```
GraphicsDevice.Clear(Color.CornflowerBlue);
```

by this line:

```
GraphicsDevice.Clear(background);
```

Run the game again and you'll see a red background. Congratulations on your first real usage of a variable!

4.4.2 Creating Your Own Colors

But colors such as `Color.Red` are just the colors that MonoGame has already defined for us. However, nothing's stopping you from creating your *own* colors that MonoGame hasn't created yet.

To create objects of a more complicated type (such as `Color` or `GameTime` or any type that you're going to create yourself), you use the keyword **new**. You've seen this word in some of our examples before, but we haven't really talked about it yet. The **new** keyword calls the **constructor** method of the object you're going to create. A constructor method is the first method of an object that gets called when you create that object. Constructors can contain parameters as well, depending on what kind of information an object needs when you create it. Just like with ordinary method calls, you write these parameters between parentheses, and separated by commas.

When you create a new `Color`, you need to specify *three* parameters: the R (red), G (green), and B (blue) components of the color you want to create. Therefore, the constructor of a `Color` has three parameters. These correspond to the R, G, and B values, in that order. Each of these three values must

be an integer between 0 and 255, where 0 means that the color component is not included at all, and 255 means that the component is used fully.

If you replace your background line by this:

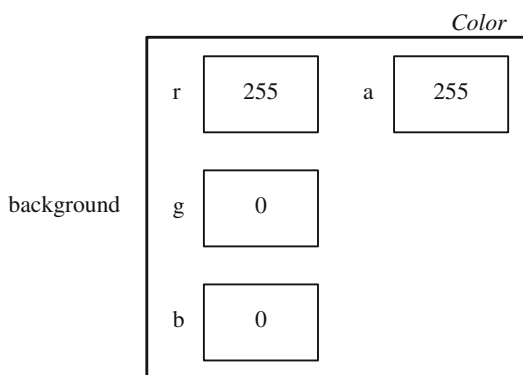
```
Color background = new Color(255, 0, 0);
```

you'll create a color that is completely red. (The expression `Color.Red` is actually shorthand for this exact same instruction, because the color red is used so often.)

If you replace the 255 by 0, you'll create the color black. If you choose a number such as 128, you'll create a color that lies between black and red (in other words, a darker shade of red). You can create the color white by setting all values to 255.

There are countless other colors that you can create.⁵ Feel free to play with these three parameters right now, to see what kinds of colors you can come up with. This so-called **RGB color model** is used worldwide for images and websites, so you can find lots of information about it online.

Assuming you've chosen the color red, the memory reserved for the background variable will look like this:



In this book, when we visualize variables in memory, we always write the *name* of the variable to the left of the memory block. If the variable has a primitive type such as `int`, then we draw the memory as a single block with a value. For more complicated class types (such as in this `Color` example), we draw the memory as a larger block that contains smaller blocks for each sub-value. We then also write the data type above this large block.

This image shows that the `Color` type actually consists of *four* sub-values. Three of them are the R, G, and B color components that you specified in the constructor. The fourth value is the A (alpha) value, which indicates the transparency: 0 means that the color is fully transparent, and 255 means that it's not transparent at all. When you create a `Color` with three parameters, the alpha component will automatically be set to 255 (indicating no transparency). But there's also a version of the `Color` constructor that takes a fourth parameter, which allows you to set the transparency yourself.

4.4.3 Using the *GameTime* Class to Change the Color

You're almost ready to let the background color change over time. Let's try to make the color change *from black to red*. What do you think we'd need to do to make this happen?

⁵Unless you feel like counting to 256^3 , which is 16,777,216.

The answer is to use a *variable* for the R(ed) component of background, instead of a fixed value. By letting this variable change from 0 to 255 over time (with a different value in each frame of the game loop), the player will see a gradually changing color.

To achieve this, you're going to use the game time. You may remember from Chap. 3 that the `GameTime` class contains information about the time that has passed in the game. It turns out that the `Draw` method has a parameter named `gameTime`, which has the type `GameTime`. (Here, `gameTime` is again a name that you can freely change yourself, but `GameTime` is the type of the variable, so you shouldn't change that one.)

When the instructions inside the `Draw` method are executed, the current game time is stored somewhere in memory, and you access it via the name `gameTime`. So, inside the `Draw` method, `gameTime` is an expression of type `GameTime`, and you can use it just like how you'd use your own variables. If you write `gameTime` somewhere in the `Draw` method, then Visual Studio will give you lots of suggestions for things you can do with this expression.

Just like the `Color` data type, the `GameTime` data type is more complicated than `int`. It consists of several integer values, such as the number of seconds passed, the number of milliseconds passed, and so on. For instance, the following expression gives you the milliseconds fraction of the total time passed since the game has started:

```
gameTime.TotalGameTime.Milliseconds
```

For example, if 13 min, 12 s, and 345 ms have passed since the game has started, the result of this expression will be 345. There is also a way to obtain the *total* passed game time in milliseconds, which would be 792,345 ms in this example $((13 \cdot 60 + 12) \cdot 1000 + 345)$. You could get this value with the following expression:

```
gameTime.TotalGameTime.TotalMilliseconds
```

Technically, `TotalGameTime` is a **property** of the class `GameTime`. You'll learn all about C# properties in Chap. 7. For now, all you need to know is that `gameTime.TotalGameTime` results in an expression of type `TimeSpan`, which is another type that exists in `MonoGame`. In turn, `TimeSpan` has a property called `Milliseconds`, which finally gives you an expression of type `int`. So you can store the result in an `int` variable like this:

```
int redComponent = gameTime.TotalGameTime.Milliseconds;
```

where `redComponent` is just a name we've chosen for now. Go ahead and add this line to the beginning of the `Draw` method.

As you may have guessed, the idea for `DiscoWorld` is to use the value of the `redComponent` variable as the R component of the background color. So, if you replace this line:

```
background = new Color(255, 0, 0);
```

with this line:

```
background = new Color(redComponent, 0, 0);
```

you will have a red component that's slightly different in each frame of the game loop. Recall that the `Update` and `Draw` methods are executed 60 times per second. This means that the milliseconds component of the total game time will gradually increase (from 0 to 999) throughout these 60 frames. In the first frame after that, a full second has passed, so the seconds counter will be one higher and the milliseconds counter will be back at zero again.

So, by giving your background color a red component that increases in each frame (until it jumps back to 0), it will look like the color gradually transforms from black to red (until it jumps back to black and the process starts over).

If you run the game by pressing F5, you'll finally see the disco world that you've been working towards. Nice work!

By the way, we claimed earlier that the components of a `Color` should lie between 0 and 255. However, the `redComponent` variable can have values between 0 and 999. Oops! Luckily for us, the creators of MonoGame have built something extra into the `Color` constructor: if any of the parameters is larger than 255, it will simply be treated as 255. For example, the expression `new Color(835, 0, 0)` actually results in a color with RGB values of 255, 0, and 0. This is why the color of `DiscoWorld` stays red for a while before jumping to black: the R component of background stays at 255 for about three quarters of a second.

If you want to change this behavior, you could (for example) divide the value of `redComponent` by 4. That way, `redComponent` will never become larger than 249 (because $999 / 4$ will be rounded down to 249), and the color will never stay red. Feel free to try it yourself by adding `/ 4` to the code in the correct place. Do you notice a side effect? Indeed, the fading from black to red becomes slower, because the `redComponent` variable increases four times as slowly.

4.4.4 The Scope of a Variable

The game is technically finished now. But we said earlier that the `Update` method will be the place where you're going to compute the background color. From a programming point of view, it's nicer to let `Update` do most of the complicated calculations and to let `Draw` simply draw the results. The `Draw` method is now a bit "cluttered" with code that doesn't immediately draw something.

So, let's take our two new lines from `Draw` (the line that creates the `redComponent` variable and the line that updates the background variable) and move them to the `Update` method. Place the lines just below the comment that says `// TODO: Add your update logic here.`

If you try to run the game again, the compiler will suddenly give an error! It will tell you that *The name "background" does not exist in the current context* (or something similar). How can this be? We've only moved some code to a different place, right?

Well, every variable in a C# program has a particular **scope**. The scope of a variable is the part of your program where you're allowed to *use* that variable. In other words, it's the area in your code where the compiler knows what you mean if you write the variable's name.

Right now, you're declaring a variable `background` in the `Update` method. If you create a variable inside a method, the scope of your variable will be *that method only*. For your `background` variable, this means the following:

- Every time your game enters the `Update` method, it will create a new variable named `background`.
- As soon as your game has executed the `Update` method (and continues with the game loop), the `background` variable will be removed from memory again. We then say that the variable "goes out of scope."
- If you try to use the `background` variable anywhere else in your program (such as in `Draw`), the compiler will not know what you mean.

A variable of this kind is also called a **local variable**. The error message that the compiler gave makes more sense now: the variable `background` doesn't exist in the `Draw` method anymore, because we've moved it to `Update`, and it only exists inside `Update`.

You *could* add the line `Color background;` somewhere in the `Draw` method again, but that doesn't do what you want in this case. That would actually create a *new* variable that exists only within the `Draw` method. You'd then have two variables called `background`, each in their own method, and they would be totally unrelated.

What we *really* want is to set a variable in one method and then use the same variable somewhere else. To achieve that, you need to give the `background` variable a *different scope*.

4.4.5 Creating a Member Variable

Near the top of the class, just below the line `SpriteBatch spriteBatch;`, add this line:

```
Color background;
```

This creates a variable named `background`, but this time it has the *entire class* `Game1` as its scope. So, we can use this variable in every method of the `Game1` class. A variable that has an entire class as its scope is called a **member variable** of that class.

Currently, this new member variable isn't used anywhere yet. To assign a value to it, go to the `Update` method and replace the following line:

```
Color background = new Color(redComponent, 0, 0);
```

by this line:

```
background = new Color(redComponent, 0, 0);
```

As you can see, you've only removed the word `Color` from the beginning of the line, but that's a pretty crucial difference. In the `Update` method, you're now storing your color in the member variable of the entire `Game1` class. (Previously, you stored it in an extra variable that only existed inside the `Update` method itself.) So now, we have a single variable `background` that the entire class can use: `Update` gives it the correct value, and `Draw` can retrieve that value again. As a result, the error in the `Draw` method has now disappeared!

You should now be able to compile and run the program again. If you play the game, it'll still do the same as before, but the program itself is nicer because we've divided the work between `Update` and `Draw`.

Why Is Scope Useful? — The concept of *scope* can be pretty difficult to understand at first. Why aren't you just allowed to declare variables in one method and use them in another method? Many new programmers struggle with this, so if you recognize this confusion, you're not alone.

An important reason why scope exists has to do with memory. In the `Update` method, we've used a variable `int redComponent` as a "helper" for computing a color. The `redComponent` variable is really only useful for that part of the code. As soon as we've computed an actual color, we don't need the `redComponent` variable anymore. By declaring it inside the `Update` method, we allow the program to clean up this variable as soon as we leave the `Update` method. We can then use the memory for something else again.

By contrast, if we would turn all variables into member variables, the program would always keep remembering a lot of temporary things that aren't useful anymore.

Let's summarize what you now know about the scope of a variable:



Quick Reference: Variable Scope

The *scope* of a variable is the area in your code where that variable can be recognized.

- If you declare a variable inside a method, you can only use it inside that method. This is called a *local variable*. When your program has finished executing the method, the variable goes “out of scope,” and the reserved memory can be reused for something else.
- If you declare a variable at class body level (so outside any of the methods of your class), you can use it in all methods of your class. This is called a *member variable*.

In game programming, you use member variables to store the game world. (In *DiscoWorld*, the game world consists only of a color.) It's common to assign values to these member variables in the *Update* method and use those values again in the *Draw* method. If you only need a variable temporarily (as a “helper” for computing something else), you use local variables.

Actually, to be complete, the scope of a variable is the smallest chunk of code between two curly braces ({ and }) that contains that variable. In this example, *background* is defined between the curly braces of the entire class (so it can be used by the whole class), and *redComponent* is defined between the curly braces of a single method (so it can be used inside that method only). Later in the book, we'll see more different examples of variable scope.

Fields and Members — In C#, a member variable is officially called a **field**. But if you ask us, that's a pretty confusing word, and the term “member variable” covers the meaning much better. The term “member variable” is also used in many other programming languages, such as Java and C++.

Actually, C# officially uses the word **member** for everything that describes the data and behavior of a class. This includes fields, methods, and other things that we haven't talked about yet. This makes the choice for the word “field” even more confusing: it's a variable *and* it's a member, so why not call it that way?

4.4.6 Summary of the *DiscoWorld* Example

The *DiscoWorld* example is now finished! Here's an overview of all the changes that you've made to the default *MonoGame* project.

- You've added the line `Color background;` near the top of the class. This means that the *Game1* class now has a *member variable* that it can use in all of its methods.
- You've also added two lines to the *Update()* method:

```
int red = gameTime.TotalGameTime.Milliseconds / 4;  
background = new Color(red, 0, 0);
```

The first line takes the millisecond component of the game time that has passed (which is a number between 0 and 999), then scales it down to fit in the range that makes sense for a color, and finally stores the result in a local **int** variable. The second line uses this variable as the red component of a new **Color** object and stores this color in the background member variable.

- And last but not least, in the **Draw** method, you've replaced **Color.CornflowerBlue** with **background**. This means that you now use your own background member variable as a parameter for the **GraphicsDevice.Clear** method. This way, the game will continuously draw the latest color that *Update* method has computed.

All games in this book will follow the pattern you've learned just now. The game world will be stored in member variables of the game's class. The **Update** method will update these variables over time, and the **Draw** method will repeatedly show what the game world currently looks like.

Listing 4.1 shows our own version of the **DiscoWorld** code. Just like **BasicGame**, it's a shortened version of the game you've created yourself, so that it fits in a single file (and on one book page). We've also removed the comments that **MonoGame** adds by default, and we've renamed the class to **DiscoWorld** (instead of **Game1**). But the code that makes the game disco-worthy is exactly the same as described in this chapter.

Listing 4.1 A program that displays a changing background color

```

1  using Microsoft.Xna.Framework;
2  using Microsoft.Xna.Framework.Graphics;
3  using System;
4
5  class DiscoWorld : Game
6  {
7      GraphicsDeviceManager graphics;
8      Color background;
9
10     [STAThread]
11     static void Main()
12     {
13         DiscoWorld game = new DiscoWorld();
14         game.Run();
15     }
16
17     public DiscoWorld()
18     {
19         graphics = new GraphicsDeviceManager(this);
20     }
21
22     protected override void LoadContent()
23     {
24     }
25
26     protected override void Update(GameTime gameTime)
27     {
28         int redComponent = gameTime.TotalGameTime.Milliseconds / 4;
29         background = new Color(redComponent, 0, 0);
30     }
31
32     protected override void Draw(GameTime gameTime)
33     {
34         GraphicsDevice.Clear(background);
35     }
36 }
```

4.5 What You Have Learned

In this chapter, you have learned:

- how to use *variables* to store information in the computer's memory;
- what a (*data*) *type* is and what the basic numerical types are in C#;
- that all *expressions* have a type and a value;
- that every variable has a certain *scope*, which can be either a single method (for local variables) or an entire class (for member variables);
- that a class is a blueprint for an object (also called an instance of that class) and can therefore be used as a type;
- how to use the Update method to change the game world via member variables and the Draw method to display the results on the screen;
- how to use the Color data type to create different colors in MonoGame.

4.6 Exercises

1. *Names*

In mathematics and physics, it's quite common to use fixed symbols for variables and constants. Can you name a few examples? What is the advantage of using these standard names and symbols?

2. *Concepts*

Provide short definitions of the concepts "instruction," "variable," "method," and "object." Which two relations does the concept "class" have with these concepts?

3. *Declaration, Instruction, Expression*

What's the difference between a declaration, an instruction, and an expression?

4. *Statement Versus Instruction*

Many programming books use the word "statement" to indicate an instruction in a programming language. Why do you think we avoid that word in this book?

5. *Changing Names*

Look at the class `DiscoWorld` in Listing 4.1. Let's say you want to change the name of this class into `TechnoWorld` (because your boss has said that disco is for old people). What would you have to change in the code then? And apart from these necessary changes, what is logical (but not required) to change as well?

6. *Playing with Colors*

In the `DiscoWorld` example, you've seen the `Color` data type, and its constructor with three parameters (for the R, G, and B components).

- a. Knowing this, what colors do the following variables represent?

```
Color color1 = new Color(255, 0, 0);  
Color color2 = new Color(0, 255, 0);  
Color color3 = new Color(0, 0, 255);  
Color color4 = new Color(255, 255, 0);
```



```

Color color5 = new Color(255, 0, 255);
Color color6 = new Color(0, 255, 255);
Color color7 = new Color(255, 255, 255);
Color color8 = new Color(255, 160, 0);
Color color9 = new Color(180, 180, 180);

```

- Change the DiscoWorld program so that the color changes from black to blue instead of from black to red.
- Now modify the program so that the color changes from black to purple.
- Can you also modify the program so that the color changes from *purple to black*?

7. Syntactical Categories

Indicate for each of the following program fragments to which syntactical category it belongs: (M)ethod call, (D)eclaration, (E)xpression, (I)nstruction, and (A)ssignment. There may be multiple correct answers for each. Watch out: some fragments are not valid code at all, and they have zero answers!

<code>int x;</code>	<code>int 23;</code>	<code>(y+1)*x</code>	<code>new Color(0,0,0)</code>
<code>(int)x</code>	<code>23</code>	<code>(x+y)(x-1)</code>	<code>new Color black;</code>
<code>int(x)</code>	<code>23x0</code>	<code>x+1=y+1;</code>	<code>Color blue;</code>
<code>int x</code>	<code>x=23;</code>	<code>x=y+1;</code>	<code>GraphicsDevice.Clear(Color.White);</code>
<code>int x, double y;</code>	<code>"x=23;"</code>	<code>spriteBatch.Begin();</code>	<code>Content.RootDirectory = "Content";</code>
<code>int x, y;</code>	<code>x23</code>	<code>Math.Sqrt(23)</code>	<code>Color.CornflowerBlue</code>
<code>""</code>	<code>0x23</code>	<code>"\\"</code>	<code>Color.CornflowerBlue.ToString()</code>
<code>"\"</code>	<code>23%x</code>	<code>(x%23)</code>	<code>game.Run()</code>
<code>"//"</code>	<code>x/*23*/</code>	<code>""</code>	<code>23=x;</code>

8. Relation Between Syntactical Categories

Have another look at your answers for the previous exercise. Do you see any patterns?

- Which combinations of categories are possible for a fragment?
- Which of the categories always belong together?
- Which of the categories always have their code fragment ending with a semicolon? Which categories have this sometimes? And which never?

9. Variable Assignment

Consider the following two variable declarations and assignments:

```

int x, y;
x = 40;
y = 12;

```

Indicate for each of the following groups of instructions what the values of x and y are when these instructions are executed after the above declarations and instructions.

<code>y = x+1;</code>	<code>x = y;</code>	<code>x = y+1;</code>	<code>x = x+y;</code>	<code>y = x/3;</code>	<code>y = 2/3*x;</code>	<code>y = x%6;</code>
<code>x = y+1;</code>	<code>y = x;</code>	<code>y = x-1;</code>	<code>y = x-y;</code>	<code>x = y*3;</code>	<code>x = 2*x/3;</code>	<code>x = x/6;</code>
			<code>x = x-y;</code>			

In one of the cases, the values of x and y are swapped. Does this work for all possible values of x and y? If so, why is that? If not, in what cases does it fail?

10. *Multiplying and Dividing*

Is there a difference between the following three instructions?

```
position = 300 - 3*time / 2;  
position = 300 - 3/2 * time;  
position = 300 - time/2 * 3;
```

11. ** Hours, Minutes, Seconds*

Suppose that the integer variable `time` contains a (possibly large) number of seconds. Write down a number of instructions that declares three new integer variables—`hours`, `minutes`, and `seconds`, corresponding to their meaning—and then fills these variables with the appropriate values. That is, `hours` should contain the number of whole hours contained in the `time` variable, `minutes` should contain the remaining number of whole minutes, and `seconds` should contain the remaining number of seconds. Note: the values of `minutes` and `seconds` should be smaller than 60. *Hint:* Use the `%` operator.

Next, write an instruction that performs the reverse operation. So, given three integer variables `hours`, `minutes`, and `seconds`, calculate the total number of seconds and store this in a new integer variable named `time`.

Part II

Game Objects and Interaction

Introduction



In this part of the book, you’re going to develop a game called *Painter*. While developing this game, you’ll learn about a lot of useful programming techniques, such as:

- writing your own methods that you can reuse (without having to copy and paste your code everywhere);
- organizing your code into multiple classes (one for each kind of object in the game);
- conditional instructions (e.g., to let something happen *if* the player presses a key);
- looping instructions (e.g., to draw an image multiple times, depending on how many lives the player has left);

For MonoGame specifically, you’ll also learn how to use game assets such as sprites and sounds and how to draw sprites at certain positions using the `Vector2` class.

About the Game: The goal of the *Painter* game is to collect paint of three different colors: red, green, and blue. The paint falls from the sky in cans that are kept floating by balloons, and you (as a player) must make sure that each can has the right color before it falls through the bottom of the screen.

You can change the color of a paint can by shooting a ball at it. By pressing the R, G, and B keys on the keyboard, you can choose which color to shoot. You can shoot a paint ball by left-clicking in the game screen. By clicking further away from the paint cannon, you give the ball a higher velocity. The place where you click also determines the direction in which the cannon shoots.

For each can that lands in the correct bin, you get 10 points. For each wrongly colored can, you lose a life. The remaining number of lives is shown in the top-left corner of the screen.

You can run the final version of this game by opening the solution belonging to Chap. 11. Press F5 and you can immediately start playing.

Chapter 5

Showing What the Player Is Doing



In the previous chapter, you’ve seen how you can use *member variables* to store the game world in a class. The Update method changes the values of these variables, and the Draw method draws the results on the screen. You’ve used this knowledge to create a game with a changing background color.

In this chapter, you’re going to create a nicer-looking game that responds to the player’s mouse movement. First, you’ll learn how to use images (also called *sprites*) and sounds in MonoGame. Next, you’ll learn about how to draw sprites at a certain position, by using the `Vector2` struct of MonoGame. This all leads to an example game in which a balloon sprite follows the mouse pointer, and a cannon barrel rotates towards the mouse pointer. The rotating cannon barrel is the first important feature of the Painter game.

5.1 Using Game Assets in MonoGame

This section explains how you can use **game assets** in MonoGame. The term “game asset” is used for all kinds of resources that your game uses, such as images, sounds, and fonts. Images in a 2D game are often also called **sprites**.

Game assets are not part of the C# code of your game. Instead, they are loaded from files that are stored somewhere else. So, if you want to create nice-looking games with images, you’ll have to store these images somewhere, tell the program to *load* these images at the right moment, and eventually *show* the images. This introduces a number of things that you need to think about:

- From which locations can you load sprites?
- How do you retrieve the information from an image file?
- How do you draw a sprite on the screen?

This section answers these questions. At the end of this section, we’ll look at music and sound effects, which are other game assets that work almost the same as sprites. Note: this section is *very* specific to MonoGame. Each game engine handles assets a bit differently. By the way, handling game assets has changed quite a lot in MonoGame 3.6. So if you search the Internet for examples, watch out for examples that are still based on an older MonoGame version.

Creating Sprites — The name “sprite” comes from *spriting*, which is the process of creating two-dimensional, partially transparent raster graphics that are used for video games. In the early days, creating these two-dimensional images was a lot of manual work. People started by drawing them on paper instead of on a screen! The process is much easier nowadays (thanks to computer programs), but artists often still draw their sprites in an old-fashioned “blocky” style on purpose. This has become an entire artistic style of its own, called *pixel art* or sprite art. (By the way, a **pixel** is a single “dot” on the screen. We’ll use that word quite a lot in this book!)

Even if you’re not an artist, it’s still useful if you can make sprites yourself. This allows you to quickly create prototypes, and maybe you’ll even discover a new talent!

To create sprites, you first of all need good tools. Most artists use painting programs like Adobe Photoshop or vector drawing programs like Adobe Illustrator. But it’s also possible to work with free tools such as Microsoft Paint or GIMP. Every tool requires practice, so work your way through some tutorials to get some insight in how a program works, and then simply draw a lot.

If you want to create a realistic-looking game, it might be useful to draw large images and then scale them down later. Save the original (unscaled) files on your computer as well! If you want to go for a pixel art style, it’s better to draw your sprites in the sizes you’ll also use in the game.

We won’t teach you anything else about this: programming and art are two completely different lines of work. If you really want to learn game programming, be careful not to lose yourself in creating great-looking artwork. Both aspects can be fun, of course, but it’s better if you don’t try to master everything at the same time.

5.1.1 Managing Assets with the Pipeline Tool

For this section, we’re going to look at the SpriteDrawing example project. You can load the project from this chapter’s supplementary files.

If you look at this project in the Solution Explorer in Visual Studio (see also Fig. 5.1), you see that it contains a folder called *Content*. Inside this folder is a single file, *Content.mgcb*. The extension “mgcb” stands for “MonoGame Content Builder,” which is the name of a helper program that MonoGame uses. The job of this program is to *build* your assets into a format that your game understands. It works very much like a compiler, but then for assets instead of C# code.

Right-click on the file *Content.mgcb*, choose “Open With...”, and select **MonoGame Pipeline Tool** from the list of options. The MonoGame Pipeline Tool is a nice visual tool in which you’ll manage all your game assets. We’ll call it the Pipeline Tool to keep things short.

When you inspect *Content.mgcb* in the Pipeline Tool (see also Fig. 5.2), you’ll see the names of two files: *snd_music.wma* and *spr_lives.png*. The first file contains background music for our game, and the second file is a sprite of a yellow balloon. These are references to real files on your computer; in this case, they are examples that we’ve created for you.

If you want, you can add your own assets by choosing Edit → Add → Existing Item... and browsing your drive for an image or sound that you like. You’ll then see a pop-up that asks you whether you want to *copy* the file to the “Content” folder or use a *link* to the file. We recommend copying the file because this will make the asset easier to use later on. For this project, though, you don’t have to add your own assets: just stick to the two files we’ve provided.

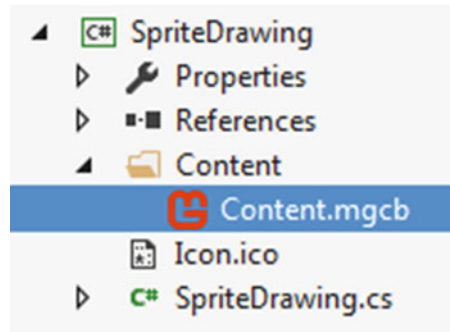


Fig. 5.1 The SpriteDrawing project in the Solution Explorer, with the *Content.mgcb* file selected

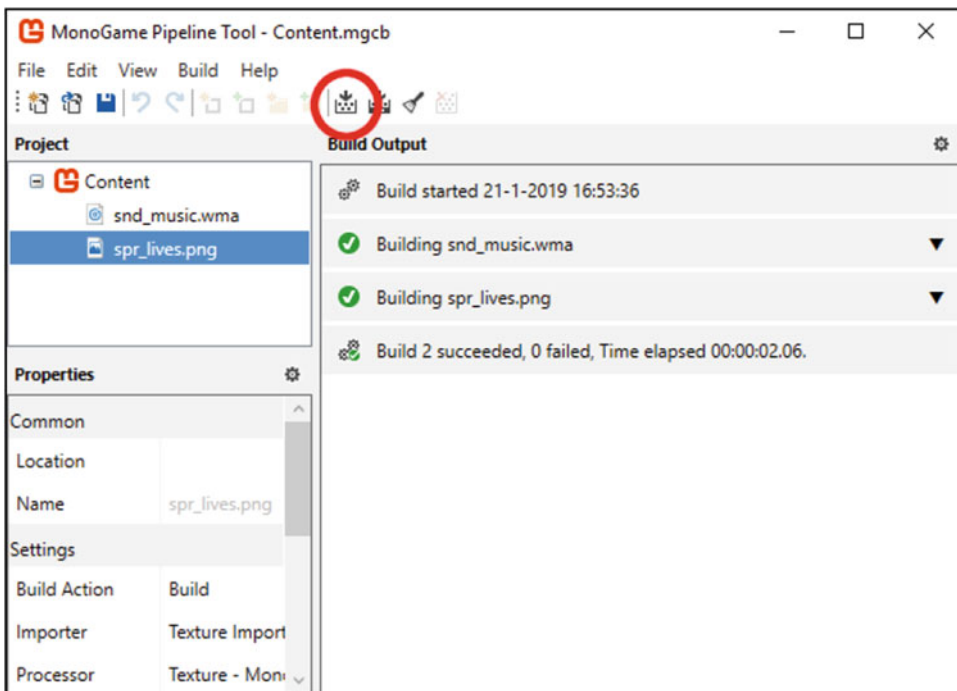


Fig. 5.2 The MonoGame Pipeline Tool, showing the *Content.mgcb* file of the SpriteDrawing project. We’ve highlighted the “Build” icon

Click the “Build” icon (the circled icon in Fig. 5.2) and MonoGame will process the assets so that you can use them in your game. You should see green icons on the right side of the Pipeline Tool window, one for each asset that was successfully built. (If this doesn’t work for you, then there may be something wrong with your MonoGame installation. In that case, please go to the book’s website for more information.)

If you want to know more about the possibilities of the Pipeline Tool, please have a look at the MonoGame documentation. Asset handling may change again in future versions of MonoGame, so their own documentation is the most reliable source of information.

Assets in Older Versions — In MonoGame 3.5 and earlier (including XNA, the predecessor of MonoGame), you could simply drag your image and sound files into the “Content” folder in Visual Studio. This was easier than it is now, but it was also a bit confusing. For example, you had to change certain properties of your images in Visual Studio, and you had to pre-process your sound files into so-called “.xnb” files. Nowadays, the Pipeline Tool manages these things for you.

The Pipeline Tool has existed for a while, but MonoGame 3.6 is the first version in which the *old* style doesn’t work anymore. So, from now on, you have no choice but to use the Pipeline tool—and although it does take some getting used to, it definitely has its advantages.

5.1.2 Telling MonoGame Where to Load Your Assets

Now, let’s use these resources in a game. The source code of the SpriteDrawing program is shown in Listing 5.1. On line 21, you’ll find the code that tells the compiler in what directory the game assets are located:

```
Content.RootDirectory = "Content";
```

This line is automatically added to every new MonoGame project that you create, so you can just leave it there. But in case you’re wondering, this is an instruction that accesses a **property** called Content, which is part of the Game class of MonoGame. (Remember that SpriteDrawing is a specific version of the Game class, as written on line 6.)

Properties are a very nice concept in the C# language, and they’re one of the main features of C# that you cannot find in Java (which is a very similar programming language in general). We’ll talk much more about properties in Chap. 7. For now, you could see a property as a sort of member variable. It’s really something different, but you can use it almost in the same way.

Anyway, the Content property contains another property called RootDirectory, and you can assign a *text* value to it. In this case, the text is the name of the content folder. In C#, we use the data type **string** for pieces of text. A string is always enclosed by double quotation marks, as you can see in the above instruction.

Listing 5.1 A program that displays a sprite on a white background

```

1 using Microsoft.Xna.Framework;
2 using Microsoft.Xna.Framework.Graphics;
3 using Microsoft.Xna.Framework.Media;
4 using System;
5
6 class SpriteDrawing : Game
7 {
8     GraphicsDeviceManager graphics;
9     SpriteBatch spriteBatch;
10    Texture2D balloon;
11
12    [STAThread]
13    static void Main()
```

```

14 {
15     SpriteDrawing game = new SpriteDrawing();
16     game.Run();
17 }
18
19 public SpriteDrawing()
20 {
21     Content.RootDirectory = "Content";
22     graphics = new GraphicsDeviceManager(this);
23 }
24
25 protected override void LoadContent()
26 {
27     spriteBatch = new SpriteBatch(GraphicsDevice);
28     balloon = Content.Load<Texture2D>("spr_lives");
29
30     MediaPlayer.Play(Content.Load<Song>("snd_music"));
31 }
32
33 protected override void Draw(GameTime gameTime)
34 {
35     GraphicsDevice.Clear(Color.White);
36     spriteBatch.Begin();
37     spriteBatch.Draw(balloon, Vector2.Zero, Color.White);
38     spriteBatch.End();
39 }
40 }

```

5.1.3 Loading Sprites

Now that you've indicated where the game assets are located, you can use MonoGame's classes and methods to load a sprite and then display it. These classes are available in the library `Microsoft.Xna.Framework.Graphics` which you import with a **using** statement at the beginning of the program:

```
using Microsoft.Xna.Framework.Graphics;
```

Let's look at how to load a sprite from a file and store it somewhere in memory, using a variable. In this example, we use a *member variable*, as you can see on line 10:

```
Texture2D balloon;
```

Just like how `Color` is a special data type in MonoGame for colors, `Texture2D` is a MonoGame data type for sprites. So, the `balloon` variable represents a place in memory that can store a sprite. Furthermore, because `balloon` is a member variable, we can use it in multiple methods of our class. We give this member variable a value (in other words, we *initialize* it) in the `LoadContent` method, on line 28:

```
balloon = Content.Load<Texture2D>("spr_lives");
```

The method `Load<Texture2D>` is part of the `Content` class that all versions of Game can use for free. This method loads a sprite from a file and then assigns it to a variable of type `Texture2D`. So, after this instruction, the `balloon` variable finally contains our sprite!

As you can see, you don't have to provide the file name extension of the sprite (.png). This is because the Pipeline Tool has already converted the image to a special file that MonoGame

understands—at this point, it no longer matters whether the image was a .png, or a .jpg, or any other file type.

When to Load Assets? — You may remember from Chap. 3 that `LoadContent` is the first place in a MonoGame program where you can safely load your assets. We’ve also explained that you don’t want to reload your assets in every frame of the game loop, because that would affect your game’s performance.

However, you don’t have to load *all* assets inside the `LoadContent` method. Assets use memory from the moment you’ve loaded them, so it’s sometimes better to only load them just before you need them in the game. For example, if you have a game with multiple levels, and each level has its own 10-min music track, it would be a bit of a waste if you loaded all music at the very beginning.

The games in this book are still relatively simple, so in our examples, we’ll load most of our assets when the game starts. But it’s useful to know that the `Content.Load` methods still work later in the program. `LoadContent` is simply the *first* opportunity for you to call them, but it’s not the *only* opportunity.

5.1.4 Drawing Sprites

We’ve loaded a sprite and stored it in a member variable now, but we still need to actually draw it on the screen. In MonoGame, drawing sprites can be done with the `SpriteBatch` class. But in order to use methods from that class, you first need a variable of the type `SpriteBatch`. Just like the graphics device (explained in Chap. 3), this is a variable that you need to initialize when the game starts. This initialization is already included in every new MonoGame project that you create, so you don’t have to worry about it yourself. Still, let’s briefly look at how it works. On line 9 of the `SpriteDrawing` example, you see another member variable:

```
SpriteBatch spriteBatch;
```

which is (as you can probably guess by now) a place in memory that can store an object of type *SpriteBatch*. We initialize this variable in the `LoadContent` method:

```
spriteBatch = new SpriteBatch(GraphicsDevice);
```

When you create a sprite batch, you need to provide the graphics device as a parameter. This is because the `SpriteBatch` object needs to know *which* graphics device it should use.

Now, let’s look at how to *use* this sprite batch in the `Draw` method to actually draw sprites. In the first instruction of the `Draw` method, we clear the screen so that any previously drawn sprites are removed:

```
GraphicsDevice.Clear(Color.White);
```

This line should be very familiar to you by now. Of course, you can choose any color here, or even use the code from the `DiscoWorld` example to let the background color vary according to the current time.

The `spriteBatch` variable stores an *object* of type `SpriteBatch`. We can use this object to call all sorts of methods from the `SpriteBatch` class. In this example, we only want to draw a single balloon sprite, so the instructions look like this:

```
spriteBatch.Begin();
spriteBatch.Draw(balloon, Vector2.Zero, Color.White);
spriteBatch.End();
```

In the `Begin` method, the graphics device is prepared for drawing things on the screen. You should call this method before you start drawing any sprites.

The `End` method tells the graphics device that you're done drawing, so that things can be cleaned up for the next drawing batch. You should call this method after you've finished drawing all sprites. Neither the `Begin` nor the `End` methods need any parameters, because there's nothing special to specify: we just start and finish drawing.

Between the `Begin` and `End` calls, you can draw as many sprites as you want. For each sprite that you want to draw, you call the `Draw` method of the `SpriteBatch` class. Note that this is a different method than the `Draw` method of the game itself. (So this game uses two methods that are both called `Draw`, but they belong to different classes.)

The `Draw` method of `SpriteBatch` takes three parameters:

- The first parameter is the sprite that needs to be drawn. This should be an expression of type `Texture2D`. In this case, we use the sprite that we've previously loaded into the `balloon` variable.
- The second parameter is the position of the sprite on the screen. This position is given as a two-dimensional coordinate vector. MonoGame uses the struct `Vector2` to represent coordinates. (Remember that a struct is almost the same as a class.) We'll talk more about `Vector2` later in this chapter.

Just like how `Color.Red` is an expression that gives you a specific color, the expression `Vector2.Zero` gives you a specific vector. This vector corresponds to the top-left corner of the game window.

- The third parameter is a color. You can use this to optionally change the tint of the sprite. If you supply the color `Color.White`, then no tinting happens, and you'll see the sprite exactly as its original image.

Figure 5.3 shows a screenshot of the program when running. Note that the position parameter indicates the *top-left corner* of the sprite. This is why you see the entire balloon: the top-left corner of the sprite lies exactly on the top-left corner of the game window.

5.1.5 Music and Sounds

Most games contain sound effects and background music. These are useful for various reasons.

Sound effects give important cues to indicate to the user that something has happened. For example, playing a click sound when the user clicks a button provides feedback to the user that the button was indeed pressed. As a second example, footstep sounds indicate that enemies might be nearby, even though the player may not see these enemies yet.

Background music and atmospheric sound effects (such as dripping water, wind in the trees, or the sound of cars in the distance) enhance the experience further. They can give the player a feeling of being present in the game world. They make the environment more alive, even when nothing is actually happening on the screen.

In MonoGame, sound files are assets (just like sprites). MonoGame makes a distinction between sound effects and background music. It even has two different data types for this: `SoundEffect` and `Song`. In

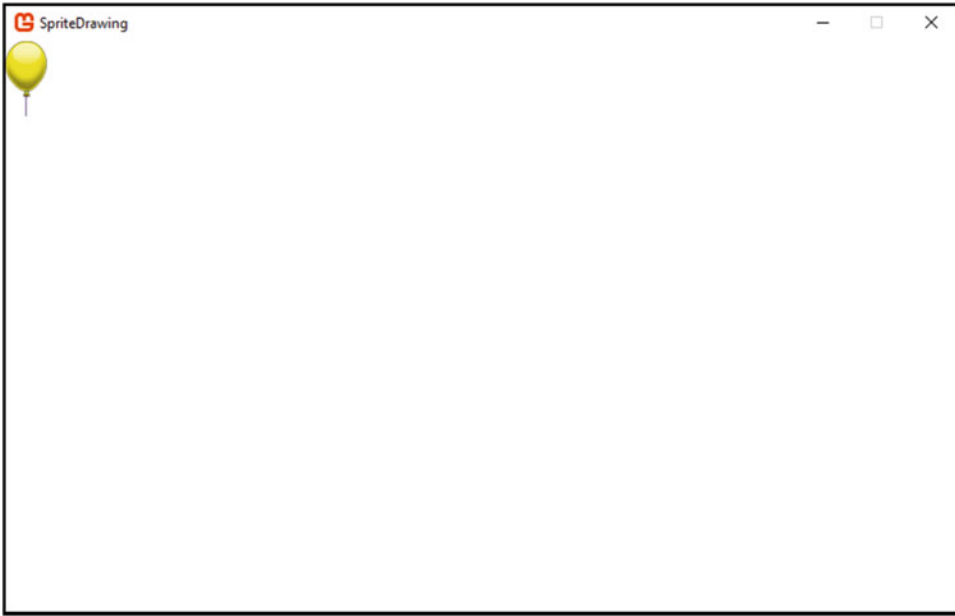


Fig. 5.3 A screenshot of the SpriteDrawing program running

the MonoGame Pipeline Tool, you need to specify for each sound file whether it is a sound effect or a song.

Open the Content.mgcb file of the SpriteDrawing project, and click on *snd_music.wma*. This file is a piece of background music that we've added for you. Elsewhere in the Pipeline Tool window, you'll see the properties of this asset, which includes the text "Song-MonoGame." This setting means that you'll be able to load the asset as a Song object in your game. If you would change this setting to "Sound Effect-MonoGame" (via the drop-down list that currently says "Song-MonoGame"), and then rebuild the assets, you'll be able to load the asset as a SoundEffect instead. But that's not what we want right now, because *snd_music.wma* is a piece of background music.

To play background music in your program, you need access to a MonoGame library named Media. This means that you need an extra **using** instruction:

```
using Microsoft.Xna.Framework.Media;
```

In the LoadContent method, you can then load the sound file and play it as background music:

```
MediaPlayer.Play(Content.Load<Song>("snd_music"));
```

The Content.Load<Song> method works exactly the same as Content.Load<Texture2D>, but it returns a Song object. (Again, you leave out the .wma extension because MonoGame has already pre-processed the song into a special file type.) So this method gives us an expression of type Song.

We could choose to store this in a member variable (just like balloon), but that's not necessary here. In this game, we just want to play the song immediately, and we never need to refer to this specific song later in the code (because it will just keep playing). So, instead of storing the result in a variable, we immediately give the result as a parameter to the MediaPlayer.Play method. This method will play the song for us.

As you can see, adding background music to your game is easy. Playing a sound effect works very similarly, but with the data type `SoundEffect`. For example, if you have included a file named *scream.wav* in the Pipeline Tool and you've marked it as a sound effect, you can load it as follows:

```
SoundEffect mySound = Content.Load<SoundEffect>("scream");
```

where `mySound` is a variable that will then store the sound effect as an object. You can then play the sound effect by calling the `Play` method on that object:

```
mySound.Play();
```

By the way, sound effects are part of a different MonoGame library which is called `Audio`, so you'll need one extra **using** instruction for this:

```
using Microsoft.Xna.Framework.Audio;
```

5.2 A Sprite Following the Mouse Pointer

Now that you know how to use sprites and sounds in MonoGame, this section lets you create a game in which the balloon sprite follows the mouse pointer. We'll explain step by step how you can extend the `SpriteDrawing` example to achieve this. If you want to see the results immediately, take a look at the *Balloon1* project in the Visual Studio solution for this chapter.

5.2.1 Adding a Background

First, let's make the example more visually appealing by adding a background sprite. A background is a sprite like any other (but bigger, so that it fills the entire screen). This means you have to add it to the game via the Pipeline Tool, load it via `Content.Load`, and eventually draw it.

In the Pipeline Tool, include the file *spr_background.jpg* from our examples and then build the assets again, so that the extra sprite will be included. For a reminder of how to do this, you can take a look at Sect. 5.1.1 again.

In the code, the first step is to add another member variable of type `Texture2D`, which will store the background sprite. So, at the top of the class, you need to declare another `Texture2D` variable. Because you now have two variables of this type, you can declare them both in a single line:

```
Texture2D balloon, background;
```

You load the sprite in the `LoadContent` method, just like the balloon:

```
background = Content.Load<Texture2D>("spr_background");
```

And finally, to draw the background in the `Draw` method, you simply add another `Draw` instruction right after `spriteBatch.Begin()`, like this:

```
spriteBatch.Draw(background, Vector2.Zero, Color.White);
```

As you can see, you're simply reusing concepts that we've explained before. That's the fun part of methods and parameters: by calling a method multiple times with different parameters, you can use the same method for different things!

By the way, it's important that you place the instruction that draws the background *before* the instruction that draws the balloon sprite. MonoGame will draw the sprites in the order you've specified. And because the background is an image that fills the entire screen, reversing this order would mean that the balloon gets hidden behind the background. (If you want to confirm this for yourself, try swapping these two instructions.) In other words, the background is the first sprite that you want to draw.

5.2.2 Using a *Vector2* Variable for the Position of a Sprite

Let's turn the position of the balloon sprite into a variable, just like what we did with the background color in *DiscoWorld*. This time, you'll use the *Vector2* type of MonoGame, which can be used to indicate a position on the screen. *Vector2* is a struct, just like *Color*. This means that you can use it as a data type: you can create variables of type *Vector2*, and these variables can then store specific *instances* of *Vector2*.

First, add a new member variable at the top of the class:

```
Vector2 balloonPosition;
```

Next, add an Update method that gives this variable a value:

```
protected override void Update(GameTime gameTime)
{
    balloonPosition = Vector2.Zero;
}
```

And finally, use this variable in the Draw method, by replacing *Vector.Zero* in the instruction that draws the balloon sprite:

```
spriteBatch.Draw(balloon, balloonPosition, Color.White);
```

Run this game and you should see the same result as before. But of course, the purpose of this new member variable is that you can change it over time. More specifically, you'll want to change its value in the Update method, so that the Draw method will draw it at a different position in each frame.

It's useful to understand the *Vector2* data type a bit better. Just like the *Color* struct, *Vector2* has a number of predefined expressions (such as *Vector2.Zero*), but it also has its own **constructor** method, which you can call via the keyword **new**. The constructor of *Vector2* has two parameters: one for the *x* component of the vector you want to create and one for the *y* component. For example, if you replace the line in Update by the following line:

```
balloonPosition = new Vector2(300, 100);
```

and then run the program again, the balloon will be shown 300 pixels to the right and 100 pixels lower than before. So, the *x* component of a *Vector2* indicates the (horizontal) number of pixels from the left side of the screen, and the *y* component indicates the (vertical) number of pixels from the top.

The *x* and *y* components of a *Vector2* have the data type **float**, so they can store numbers with a fractional component as well. But you can also assign **int** values to them; in that case, your program will automatically convert these values to **float** values.

But instead of choosing the same vector in every frame, you can make the game more interesting by changing the balloon's position over time. This is very much comparable to how you changed the red component of the background color in *DiscoWorld*.

In the `Update` method, you can use the expression `gameTime.TotalGameTime.Milliseconds` to get an `int` value between 0 and 999, which increases over time. You can then plug this number into the `Vector2` constructor to give `balloonPosition` dynamic coordinates. For example, try replacing the `Update` method by the following lines:

```
protected override void Update(GameTime gameTime)
{
    int yPosition = 480 - gameTime.TotalGameTime.Milliseconds / 2;
    balloonPosition = new Vector2(300, yPosition);
}
```

Can you guess what these instructions do and what the result will look like for the player? The x -coordinate of the balloon will always be 300, but the y -coordinate will change over time, as indicated by the local variable `yPosition`.

When `gameTime.TotalGameTime.Milliseconds` is zero, `yPosition` will receive a value of 480. As `gameTime.TotalGameTime.Milliseconds` grows to 999, `yPosition` will shrink to (approximately) -20 . (Try to verify these values for yourself.)

As a result, you will see a balloon that starts at the bottom of the screen and then moves up, until it jumps back to the bottom again. Try to play around with the numbers used to calculate the new balloon position to see how this works. Can you make the balloon move from left to right? Or diagonally across the screen? Can you make it move faster or slower?

CHECKPOINT: FlyingSprites



You can also find this example in this chapter's supplementary files, under the project name *FlyingSprites*. Throughout this book, you will see more "checkpoint" markers like this one. When you've reached a checkpoint, your code should match a certain example project from the book.

Helper Variables (1) — You've used a local variable `int yPosition` to store the result of a mathematical expression. But this expression itself already has the data type `int`, so we could also have placed it inside our vector immediately:

```
balloonPosition = new Vector2(300, 480 - gameTime.TotalGameTime.Milliseconds/2);
```

This is a shorter version of the code that does exactly the same. Similarly, in the *DiscoWorld* game of Chap. 4, we could have written shorter code that doesn't require a local variable `int redComponent`:

```
background = new Color(gameTime.TotalGameTime.Milliseconds/4, 0, 0);
```

However, in both cases, we chose to use a so-called *helper variable* to make the code easier to read. But it's good to remember that you don't have to create helper variables for *everything*. To some extent, it's a matter of personal taste.

5.2.3 Retrieving the Mouse Position

Now that you know how to draw a sprite at a particular position on the screen, let's go a step further by showing the balloon at the current mouse position. As the player moves the mouse over time, it will look like the balloon is “following” the mouse. But in reality, this is just an illusion that you create by repeatedly drawing the sprite at a different position.

This will be the first example of a game that uses the player's input (namely, **mouse input**) to influence what the player sees. Of course, this interaction with the player is what makes games truly fun to play (and fundamentally different from movies).

To let the balloon sprite follow the mouse, you only need to make a few changes to the program you already have now. The first thing to know is that MonoGame has a special `Mouse` class, which you can use to obtain information about the mouse. This class is part of another library that you have to include, called *Input*. So, at the top of your file, add another **using** instruction to load this library:

```
using Microsoft.Xna.Framework.Input;
```

The `Mouse` class has a method called `GetState`. This method gives you the current mouse state. Using this current mouse state, you can find out where on the screen the mouse is, whether the player is clicking on one of the buttons, and so on. The *type* of the current mouse state is `MouseState`, and you can create a *variable* of that type to store the current mouse state for later use. Comment out (or remove) the two instructions that are currently in the `Update` method. Add the following instruction instead:

```
MouseState currentState = Mouse.GetState();
```

With this variable, you can retrieve the *x*- and *y*-position of the mouse in the screen using its `X` and `Y` properties. These properties are of type `int`, so you can immediately use them in the `balloonPosition` vector, like this:

```
balloonPosition = new Vector2(currentMouseState.X, currentMouseState.Y);
```

Because you will draw the balloon sprite at that position in the `Draw` method, and both the `Update` and `Draw` methods are executed in every iteration of the game loop, the final result is that the balloon is always drawn at the current mouse position. If you press F5 and execute the program, you'll see exactly this behavior.

By the way, for this example, it's useful to draw the mouse cursor on the screen as well. MonoGame hides the cursor by default, but you can change that by adding the following line to your class constructor:

```
IsMouseVisible = true;
```

This is a feature that your game automatically “borrows” from the `Game` class. The word **true** here is probably new to you—but we'll talk much more about it in the next chapter.

The complete program so far is shown in Listing 5.2.



CHECKPOINT: Balloon1

You can also find this program in the example projects for this chapter, under the name `Balloon1`.

Helper Variables (2) — Notice how we’ve used another helper variable here: `MouseState currentMouseState`. This variable is especially useful because we use it multiple times, namely, for `currentMouseState.X` as well as for `currentMouseState.Y`. According the previous gray textbox, you could also have computed the balloon’s position *without* this helper variable:

```
balloonPosition = new Vector2(Mouse.GetState().X, Mouse.GetState().Y);
```

This is also a valid code (and it does exactly the same thing), but there’s one subtle difference: in this new version, you call the `Mouse.GetState()` method two times. This is a bit of a waste, because you know that this method will return the same expression both times. By only calling the method once, and storing the result in a helper variable, you can *reuse* the result without having to recalculate it.

Listing 5.2 A program that shows a balloon following the mouse

```

1  using Microsoft.Xna.Framework;
2  using Microsoft.Xna.Framework.Graphics;
3  using Microsoft.Xna.Framework.Input;
4  using System;
5
6  class Balloon : Game
7  {
8      GraphicsDeviceManager graphics;
9      SpriteBatch spriteBatch;
10     Texture2D balloon, background;
11     Vector2 balloonPosition;
12
13     [STAThread]
14     static void Main()
15     {
16         Balloon game = new Balloon();
17         game.Run();
18     }
19
20     public Balloon()
21     {
22         Content.RootDirectory = "Content";
23         graphics = new GraphicsDeviceManager(this);
24         IsMouseVisible = true;
25     }
26
27     protected override void LoadContent()
28     {
29         spriteBatch = new SpriteBatch(GraphicsDevice);
30         balloon = Content.Load<Texture2D>("spr_lives");
31         background = Content.Load<Texture2D>("spr_background");
32     }
33
34     protected override void Update(GameTime gameTime)
35     {
36         MouseState currentMouseState = Mouse.GetState();
37         balloonPosition = new Vector2(currentMouseState.X, currentMouseState.Y);
38     }
39
40     protected override void Draw(GameTime gameTime)
41     {

```

```
42 GraphicsDevice.Clear(Color.White);
43 spriteBatch.Begin();
44 spriteBatch.Draw(background, Vector2.Zero, Color.White);
45 spriteBatch.Draw(balloon, balloonPosition, Color.White);
46 spriteBatch.End();
47 }
48 }
```

5.3 Changing the Sprite Origin

When you run the Balloon1 example, you'll notice that the balloon is drawn such that the top-left corner of the sprite is at the current mouse position. This is the default behavior for when you draw a sprite at a given position. We also say that the top-left corner is the sprite's **origin**.

5.3.1 Using the Width and Height of a Sprite

But what if you want to *change* the origin of a sprite? For example, suppose that you would like to draw the *center* of the balloon at position `balloonPosition`. Well, the `Texture2D` class has two nice properties that you can use. Given a sprite `mySprite`, the expressions `mySprite.Width` and `mySprite.Height` give you the width and height of the sprite in pixels. These expressions have the type `int` again, so you can easily use them in your own code.

To draw the center of the balloon at the mouse position, you need to change your code so that the balloon is drawn a bit more to the left and a bit more to the top. These “bits more” should be exactly half of the sprite's width and height, respectively. So, the following line will do what you want:

```
balloonPosition = new Vector2(currentMouseState.X - balloon.Width / 2,
                             currentMouseState.Y - balloon.Height / 2);
```

Notice how the two parameters of our `Vector2` constructor are now more complicated mathematical expressions, which include the `-` and `/` operators. This is all perfectly allowed, because these expressions have a result of type `int` again. We could have used helper variables again, but in this case, we chose not to. The downside is that this instruction is now quite long: we've had to divide it over two lines, to make it fit on a page of this book.

Finally, replace the expression `balloon.Height / 2` by the expression `balloon.Height`. This will move the sprite's origin to the bottom center of the sprite. This makes the end of the balloon's rope align nicely with the mouse position, as if the player is “holding” the rope.

5.3.2 Vector Math

There's a slightly nicer way (programming-wise) to achieve the same “origin” effect. You can use another variable of type `Vector2` to store the origin of the sprite separately. Then, when you draw the sprite, you can subtract this vector from `balloonPosition` to shift the sprite.

First, revert the `balloonPosition` line to its previous state:

```
balloonPosition = new Vector2(currentMouseState.X, currentMouseState.Y);
```

Then declare a new member variable named `balloonOrigin`, of type `Vector2`:

```
Vector2 balloonOrigin;
```

(You could also declare `balloonOrigin` and `balloonPosition` on the same line.) We use a *member variable* for this because the size of the sprite will never change. So we can compute this vector once (in a method of our choice) and then we can use it in all other methods, without ever having to recompute it again. A good place to do this is in the `LoadContent` method, because that's where you load the sprite for the first time. So, add the following line at the end of that method:

```
balloonOrigin = new Vector2(balloon.Width / 2, balloon.Height);
```

When to Compute What? — You've made sure that the `balloonOrigin` vector is computed only once, in the `LoadContent` method. This touches upon a very important issue in game programming: *efficiency*. If you would calculate this vector in the `Draw` method, you'd perform this calculation 60 times per second as long as the game is running. In this example, it would probably not affect the performance of the application that much because the calculation is not that complicated. However, once your games start to become more complex, it is crucial to calculate things at the right place so that you don't waste computation power.

You can also improve efficiency by using helper variables in the right places. For example, if the `Mouse.GetState()` method from before would take a whole second to execute (which it luckily doesn't!), the `currentMouseState` helper variable would save us a second per frame, because it prevents a second method call.

These kinds of choices can strongly influence the performance of your game, so it's good to think about it while programming. Even though you won't notice the difference in smaller projects, writing efficient code is good practice in general.

Finally, in the `Draw` method, change the line that draws the balloon sprite:

```
spriteBatch.Draw(balloon, balloonPosition - balloonOrigin, Color.White);
```

Run the game again and you'll see that the balloon's rope still ends at the cursor.

You may have already noticed something interesting: the `-` operator works not only for numbers (such as `int`) but for the `Vector2` data type as well! For example, if you have two vectors `vectorA` and `vectorB`, then the following expression:

```
vectorA - vectorB
```

gives you a *new* object of type `Vector2`, which contains `vectorA.X - vectorB.X` as its *x* component and `vectorA.Y - vectorB.Y` as its *y* component. In other words, it's a shorter way of writing this:

```
new Vector2(vectorA.X - vectorB.X, vectorA.Y - vectorB.Y)
```

The `+` operator is defined similarly (and you can probably guess what it does). The `*` and `/` operators also exist for vectors, and they are convenient ways to multiply (or divide) a vector by a *number*. This

is the same as multiplying (or dividing) both the x and y components by that number. In other words, these two expressions:

```
vectorA * 2
vectorB / 10
```

are short versions of these two expressions:

```
new Vector2(vectorA.X * 2, vectorA.Y * 2)
new Vector2(vectorB.X / 10, vectorB.Y / 10)
```

You can use this so-called **vector math** to make your code much easier to understand.

Operators on More Complex Types — MonoGame has defined how you can use mathematical operators on `Vector2` objects. Here's another interesting example: you can multiply a `Color` by a number (of type `float`). This scales all components of a color by the given number.

Technically, you could define arithmetic operators for every class (or struct) that you write. This is a programming technique called **operator overloading**, which allows a programmer to (re)define the meaning of arithmetic operators when used on a certain data type.

But it doesn't always make sense, so MonoGame hasn't done this for all data types. For instance, the compiler won't accept the following line of code:

```
SpriteBatch halfASpriteBatch = spriteBatch / 2;
```

because why would you ever want to divide a `SpriteBatch` object by a number?

5.3.3 Multiple Versions of a Method

Interestingly, the `SpriteBatch` class actually defines multiple versions of the `Draw` method. These versions all have a different number of parameters. The most extensive version has a whopping *nine* parameters, each with their own different meaning. With that version, you can even include the sprite's origin as a separate parameter, so you don't have to do the vector math yourself. Here's how you would call that method in this example:

```
spriteBatch.Draw(balloon, balloonPosition, null, Color.White,
    0.0f, balloonOrigin, 1.0f, SpriteEffects.None, 0);
```

This looks pretty intimidating, so we won't go into all of these parameters. Feel free to experiment with them to see what happens.

There are two interesting parameters that are fun to play with. The seventh parameter (which currently has the value `1.0f`) indicates the *scale* of the sprite, as a `float` expression. If you make this value larger, the sprite will be "blown up" to a bigger version.¹ The fifth parameter (currently `0.0f`) lets you rotate the sprite around the origin. We'll see an example of that later in this chapter. There are more parameters in this method, and a keyword that you've not seen before (`null`), but we'll deal with that later in the book.

¹... which, if we think about it, is a pretty useful feature for a balloon.

That's it—you now understand how to draw sprites at particular places using the `Vector2` struct, how to compute new vectors out of other vectors, and how to use the extended version of `spriteBatch.Draw` to draw a sprite with special settings.

**CHECKPOINT: Balloon2**

The results so far can also be found in the `Balloon2` project of this chapter.

5.4 A Rotating Cannon Barrel

The final `Painter` game for this part of the book contains a cannon barrel that rotates according to the mouse position. You can now write the part of the program that does this, using the things you've learned so far. This section will guide you through it, but slightly faster than before, because many elements are the same as for the moving balloon sprite.

If you follow this section, you should end up with a project that's similar to the `Painter1` example project for this chapter. This project no longer contains the moving balloon sprite. Therefore, it's probably smart to start a new project, so that you don't lose the results of the previous sections.

5.4.1 More Sprites and Origins

The project should contain two sprites: the background you already know and the cannon barrel which can be found under the name `spr_cannon_barrel.png`. Make sure to include these assets in your project. Next, declare two `Texture2D` member variables that will store the two sprites, and declare two more `Vector2` variables that will store the position and origin of the cannon barrel:

```
Texture2D background, cannonBarrel;  
Vector2 barrelPosition, barrelOrigin;
```

In the `LoadContent` method, you load the sprites and you assign a value to the `barrelPosition` variable, as follows:

```
background = Content.Load<Texture2D>("spr.background");  
cannonBarrel = Content.Load<Texture2D>("spr_cannon_barrel");  
barrelPosition = new Vector2(72, 405);
```

The position of the barrel was chosen by us so that it fits nicely on the cannon base that is already drawn on the background.

The barrel image contains a circular part with the actual barrel attached to it. We would like the barrel to rotate around the center of that circular part. That means that you have to set this center as the sprite's origin. If you add the following instruction to the `LoadContent` method, the origin will be exactly at that point:

```
barrelOrigin = new Vector2(cannonBarrel.Height, cannonBarrel.Height) / 2;
```

5.4.2 An Angle Variable

So far, these lines of code are all similar to what you’ve seen before. The new part is this: we’re going to let the cannon barrel sprite **rotate** according to the mouse position. To prepare for this, add another member variable that will store the current *angle* of the barrel:

```
float angle;
```

And finally, draw the cannon barrel using the 9-parameter version of `spriteBatch.Draw` that we’ve explained earlier. After all, this extended version of `spriteBatch.Draw` allows you to rotate the sprite (via its fifth parameter). Therefore, apply the angle variable as this fifth parameter:

```
spriteBatch.Draw(cannonBarrel, barrelPosition, null, Color.White,
    angle, barrelOrigin, 1.0f, SpriteEffects.None, 0);
```

But now the angle variable doesn’t have a value yet. You’re going to calculate this value in the `Update` method, because the value can change in each frame of the game loop. In the `Update` method, the first step is to retrieve the current mouse position, just like before:

```
MouseState mouse = Mouse.GetState();
```

The expressions `mouse.X` and `mouse.Y` will now give you the *x*- and *y*-coordinates of the mouse.

5.4.3 Computing Angles with the Math Class

You still need to *use* the mouse position to actually calculate at what angle the barrel should be drawn. The angle should somehow depend on where the mouse is located relative to the cannon barrel. Figure 5.4 shows the situation. Here, “angle” represents the angle that we would like to compute.

Calculating the angle involves some math that you may remember from high school. You can calculate it using the trigonometric *tangent* function, which is given as follows:

$$\tan(\text{angle}) = \frac{\text{opposite}}{\text{adjacent}}$$

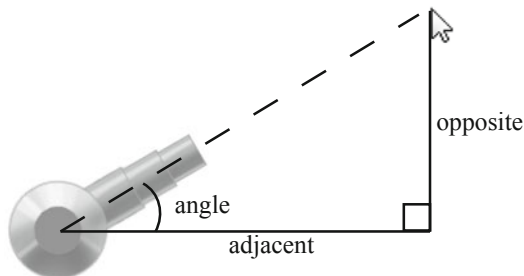


Fig. 5.4 Calculating the angle of the barrel based on the mouse position

where *opposite* and *adjacent* are the lengths of the triangle sides shown in Fig. 5.4. If you rewrite this equation a little bit, you'll see that the angle is given by the following equation:

$$angle = \arctan\left(\frac{opposite}{adjacent}\right)$$

In our game, the lengths of the opposite and adjacent sides depend on the mouse position. You can calculate these lengths by comparing the coordinates of the mouse to the coordinates of the cannon barrel. In C#, it looks like this:

```
double opposite = mouse.Y - barrelPosition.Y;
double adjacent = mouse.X - barrelPosition.X;
```

Add this code to the Update method. As an exercise for yourself, compare this code to Fig. 5.4 and verify that it's correct. So, we've calculated the opposite and adjacent sides now, but is there also a C# version of the arctan function? Fortunately, yes: there is a very helpful class called `Math` for these kinds of things. This class is contained inside the System library, so you need an extra **using** instruction:

```
using System;
```

The `Math` class contains many methods for basic mathematical calculations. If you type `Math.` somewhere in the Update method, Visual Studio will show you all of them. In particular, there are trigonometric functions like `Sin`, `Cos`, and `Tan`, as well as their inverses `Asin`, `Acos`, and `Atan`. You will probably recognize these from your math lessons (and calculators) in high school, and they do what you'd expect.

It sounds like the `Atan` method will do the job for this example. However, there is a problem with calling `Atan` in a situation where the mouse is exactly above the barrel (i.e., if `mouse.X` and `barrelPosition.X` have exactly the same value). In that case, the value of `adjacent` will be zero, and you would end up dividing by zero in the angle calculation. Dividing by zero is usually a bad idea because you can't really predict what will happen next.

Because this situation occurs pretty often, the `Math` class also has an alternative for the `Atan` method, called `Atan2`. This method takes the opposite and adjacent lengths as separate parameters, and it has a special case built in for when the adjacent length is zero. In that special case, the method will return an angle of 90°, which (in our case) will make the cannon barrel point upward, exactly as we'd like.

The following instruction sets the angle variable to the correct value:

```
angle = (float)Math.Atan2(opposite, adjacent);
```

Note that you should do a cast from **double** to **float** here. This is necessary because the `Math.Atan2` method returns a value of type **double**, but our angle variable has type **float**. Because a **float** number is less precise than a **double** number, you have to tell the compiler explicitly that you're OK with this conversion.

In case you're wondering, we could also have chosen the **double** data type for our angle member variable, and then you wouldn't to cast a **double** to a **float** here. However, the `spriteBatch.Draw` method eventually wants the angle to be given as a **float**, so then you'd have to do the same cast after all (but on a different line).



CHECKPOINT: Painter1

Run the program and enjoy the result! The barrel should now rotate to follow the mouse pointer. If you want to check your code, take a look at the Painter1 example project.

Radians and Degrees — You may remember from your math lessons that angles can be expressed in *degrees* or in *radians*. Degrees are a bit more logical for humans to understand: a full circle is 360° . But programming languages actually prefer *radians* instead: a full circle is $2 \cdot \pi$ radians (where π is a famous mathematical constant, approximately equal to 3.14). So, 360° is the same as $2 \cdot \pi$ radians.

The Math methods in C# also use radians instead of degrees. The Sin, Cos, and Tan methods expect an angle in radians as input. The Asin, Acos, Atan, and Atan2 methods will give you an angle in radians as their result. So although we said that Atan2 has a special case that returns an angle of 90° , this special case *actually* returns an value of $\pi/2$.

In general, it's worth remembering that C# uses radians. For example, if you ever want to rotate something by 1° , don't accidentally use a value of 1, because one radian is a much larger angle than 1° !

5.5 What You Have Learned

In this chapter, you have learned:

- how to include assets (such as sprites and sounds) in a MonoGame project;
- how to draw sprites and play sounds in MonoGame;
- that the constructor method of a class is responsible for creating an instance of that class;
- how to change the position of a sprite using MonoGame's Vector2 class;
- how to read the current mouse position using MonoGame's MouseState class;
- that the System library contains a Math class with helpful mathematical functions, for example, for calculating angles.

5.6 Exercises

1. Drawing Sprites in Different Locations

Modify the SpriteDrawing program so that three balloons are drawn: one in the top right corner of the screen, one in the bottom left corner of the screen, and one in the middle of the screen. *Hint:* use the Width and Height properties of the Texture2D class.

2. Flying Balloons

This programming exercise uses the FlyingSprites example as a basis.

- a. Modify the program so that the balloon flies from the top to the bottom of the screen.
- b. Now modify the program so that the balloon flies in circles around the point that indicates the center of the screen. Use the Sin or Cos methods that are available in the Math class. Define a constant that contains the speed at which the balloon turns around the center point. Also define a constant that contains the distance from the balloon to the center point (e.g., the radius of the circle).
- c. Change the program so that the balloon flies in circles around a point moving from the left to the right.

- d. Change the program so that a second balloon also turns in circles around that point, but in the opposite direction, and with a bigger radius. Feel free to try a couple of other things as well. For example, create a balloon that flies in a circle around another flying balloon or use another `Math` method to change the position of the balloon. Don't go too crazy on this or you'll end up with a headache!

Chapter 6

Reacting to Player Input



In this chapter, you will learn how your game program can react to mouse clicks and button presses. In order to do this, you need an instruction called **if** that executes an instruction (or a group of instructions) *if* a certain condition is met. These conditions are expressions of a new data type we haven't talked about yet: the **bool** data type. Furthermore, this chapter also introduces *enumerated types* as another kind of primitive type.

At the end of this chapter, you'll have extended the Painter game so that the player can change the cannon's color by pressing the R, G, and B keys.

6.1 Reacting to a Mouse Click

In the previous chapter, you've used the `MouseState` class to retrieve the mouse position. However, a `MouseState` object contains a lot of other information as well. For example, it can be used to find out whether a mouse button is currently pressed or not. For this, you can use the properties `LeftButton`, `MiddleButton`, and `RightButton`. These properties give you a value of type `ButtonState`, which you can store in a variable as follows:

```
ButtonState left = currentMouseState.LeftButton;
```

assuming that `currentMouseState` is a variable that stores the current mouse state.

Based on what you've seen so far, you might think that `ButtonState` is a class. However, it's actually a concept that we haven't run into yet: an **enumerated type**, also called an *enum* for short.

6.1.1 Enumerated Types

An enumerated type is very similar to the integer type, with the difference that instead of numeric values, the type contains words describing different states. For example, `ButtonState` is an enum with two possible states (`Pressed` and `Released`), and `MonoGame` uses it to say whether a mouse button is being pressed or not.

Enumerated types are quite handy for representing a variable that can only have a few different values, where each value has a clear meaning. As a programmer, you can also define your own enums

with their own states. For example, suppose you want to store the sort of character that a player represents. You can use an enumerated type to define these different game characters, as follows:

```
enum CharacterClan { Warrior, Wizard, Elf, Spy };
```

The **enum** keyword indicates that you are going to define an enumerated type. After that, you write the name of the enum, followed by (between curly braces) the different values that this enum can have. In this example, `CharacterClan` is now a type that you can use in your program, and an expression of type `CharacterClan` can have four possible values. Here is an example of using the `CharacterClan` enumerated type:

```
CharacterClan myClan = CharacterClan.Warrior;
```

In this case, you have created a variable of type `CharacterClan`, which may contain one of the following four values: `CharacterClan.Warrior`, `CharacterClan.Wizard`, `CharacterClan.Elf`, or `CharacterClan.Spy`. In a very similar way, the `ButtonState` type is defined somewhere in the `MonoGame` library, approximately like this:

```
enum ButtonState { Pressed, Released };
```

As another example, you can use an enumerated type to indicate the days in the week or the months in a year:

```
enum MonthType { January, February, March, April, May, June, July, August,  
                September, October, November, December };  
enum DayType { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };  
MonthType currentMonth = MonthType.February;  
DayType today = DayType.Tuesday;
```

Behind the scenes, the compiler will treat enums as ordinary numbers. The first option of an enum is typically treated as 0, the second as 1, and so on. So, technically, you can also just use an **int** to represent the same thing. However, the main advantage of enums is (of course) that programmers can assign a human-readable *meaning* to each possible number. Furthermore, enums can prevent you from accidentally using numbers that don't make sense (such as the number 42 for a month), whereas regular integers don't offer this protection.

You can place the definition of an enum inside a method, but you can also define it at the class body level, so that all the methods in the class can use that type. You can even define the enum as a *top-level* declaration (outside of the class body), and then your entire *program* can use it.

6.1.2 Executing Instructions Depending on a Condition

As a simple example of how you can use the mouse button state to do something, let's make a simple extension of the `Painter1` program from the previous chapter. This program already contains a rotating cannon barrel that follows the mouse pointer. You will now change this program so that it *only* calculates the cannon's new angle if the left mouse button is pressed. To achieve this, you'll have to change the instructions in the `Update` method, because that is where you calculate the barrel angle.

Until now, all the instructions you've written had to be executed all the time. For example, drawing the background sprite and the cannon barrel sprite always needed to happen. But this time, calculating the barrel angle only has to happen *sometimes*, namely, when the player presses the left mouse button. In broader terms, you want to execute an instruction only if a certain *condition* holds. This kind of

instruction is called a **conditional instruction**. In C# (and in many other languages), you can write conditional instructions using the keyword **if**.

With the **if** instruction, you can provide a condition and execute a block of instructions if this condition is true. (This inner block of instructions is sometimes also referred to as a **branch**, because it's a "possible path" that your program can take.) Intuitively, a condition is some statement about your program that is either true or not. Here are some (informal) examples of conditions:

1. The player has pressed the left mouse button;
2. The number of seconds passed since the start of the game is larger than 1000;
3. The balloon sprite is exactly in the middle of the screen;
4. All monsters in the game world are dead.

In C# (and in almost every other programming language), you write these conditions as *expressions*, and these expressions have a special primitive data type. This type is called the **Boolean** data type, and you use the keyword **bool** for it. An expression of type **bool** can have two possible values: **true** (if the condition holds) or **false** (if the condition does not hold). For example, if you have a variable **int** *x*, the following expression:

```
x < 200
```

has a value of **true** if *x* stores a value smaller than 200 and *false* if *x* stores a value of 200 or more. We'll look at the **bool** data type in more detail in the next section. For now, let's look at how to use Boolean expressions in an **if** instruction.

Let's say you want to execute some instructions only when the value of your variable *x* is smaller than 200. Using an **if** instruction, this looks as follows:

```
if (x < 200)
{
    // your instructions here
}
```

As you can see, an **if** instruction starts with the word **if**, followed by a Boolean expression between parentheses. (This expression is often called the *if-condition*.) After that, you write a block of instructions between curly braces. This block will only be executed if the if-condition has a value of **true** (or, intuitively, if the answer to the statement between brackets is "yes").

Here's an example of some code that you could add at the end of the `Update` method of `Painter1`:

```
if (mouse.X > 200)
{
    angle = 0;
}
```

Can you guess what this code fragment does? Recall that the variable `MouseState mouse` contains information about the mouse's current state. For instance, `mouse.X` returns the current *x*-coordinate of the mouse within the game window. So the statement `mouse.X > 200` is a Boolean expression that has the value **true** whenever the mouse is more than 200 pixels to the right of the screen's origin. Feel free to add this code and then run the game: you'll see that the cannon barrel will always be horizontal as soon as the cursor is far enough to the right.

You can also place multiple instructions between the curly braces. In that case, all these instructions will be executed only when the if-condition is true. For instance, what do you think this example would do?

```

if (mouse.X > 200)
{
    angle = 0;
    barrelPosition = Vector2.Zero;
}

```

If the mouse is far enough to the right, this code sets the cannon's angle to zero *and* changes the `barrelPosition` variable to the zero vector. This will move the cannon barrel to the top-left corner of the screen. (Note that the cannon barrel will then remain stuck there forever, because we haven't written any code that returns the barrel to its original position. Also, moving the cannon barrel somewhere else is not our actual goal. This was just a toy example to show what you can do with an `if` instruction.)

If you only want to execute a single instruction when the `if`-condition holds, you can leave out the curly braces. So, you could shorten the first angle example as follows:

```

if (mouse.X > 200)
    angle = 0;

```

The indentation of the second line is added for clarity, but it's not necessary. You can even write the whole instruction on a single line:

```

if (mouse.X > 200) angle = 0;

```

Some programmers prefer to always use curly braces and indentation, even if the `if` block contains only one instruction. This makes the code slightly longer but also easier to read: you can clearly see which instructions belong to the body of the `if` instruction, regardless of the indentation. In this book, we'll sometimes leave out the braces to keep things short.

Our real goal for this game is to update the cannon barrel angle only when the player presses the left mouse button. So, in the `Update` method, you should check whether the mouse variable currently says that the left mouse button is being held down. This is given by the following Boolean expression:

```

mouse.LeftButton == ButtonState.Pressed

```

The `==` operator compares two values and yields *true* if these values are the same and *false* otherwise. On the left-hand side of this comparison operator, you've written the expression that holds the current left mouse button state. On the right-hand side, you've written the value that you would like this state to have, namely, `ButtonState.Pressed`. So, this condition indeed checks whether the left button is currently pressed. You can now use it in an `if` instruction as follows in the `Update` method:

```

MouseState mouse = Mouse.GetState();
if (mouse.LeftButton == ButtonState.Pressed)
{
    double opposite = mouse.Y - barrelPosition.Y;
    double adjacent = mouse.X - barrelPosition.X;
    angle = (float)Math.Atan2(opposite, adjacent);
}

```

Only if the left mouse button is being held down, you calculate the angle and store its value in the angle member variable. Run the program and you'll see the effect: whenever the left mouse button is down, the cannon barrel follows the mouse pointer. Whenever the left mouse button is *not* down, the angle is not recalculated, so the cannon barrel keeps the last used angle (until you press the left mouse button again).



CHECKPOINT: Painter1a

You can also find this code in the Painter1a example program for this chapter.



Quick Reference: The if Instruction

To execute code only when a certain condition holds, write the following:

```
if (myCondition)
{
    ...
}
```

where `myCondition` can be any expression of type **bool**, and ... should of course be replaced by your code of choice. The code inside the curly braces will only be executed if `myCondition` has the value **true**.

If you only want to execute one instruction when `myCondition` holds, you can omit the curly braces.

6.2 Boolean Expressions

The condition in the header of an **if** instruction is an expression of type **bool**, which is also called a *Boolean expression*.¹ This section discusses Boolean expressions and the **bool** data type in more detail.

As said before, a Boolean expression can only have two possible values: **true** (if the expression is indeed the truth) or **false** (if the expression is not the truth). You can use such an expression in an **if** instruction to execute certain parts of your code only when certain conditions hold.

The data type **bool** is a primitive type, just like **int** or **float**. Because an expression of type **bool** can only have two possible values, it uses only 1 bit of computer memory.

6.2.1 Comparison Operators

You've already seen examples of Boolean expressions that include symbols like `<`, `>`, and `==`. These are so-called **comparison operators** that you can use to check how two expressions relate to each other. There are several comparison operators that you can use in Boolean expressions:

- `<` smaller than
- `<=` smaller than or equal to
- `>` larger than
- `>=` larger than or equal to
- `==` equal to
- `!=` not equal to

¹ Boolean expressions are named after the English mathematician and philosopher George Boole (1815–1864).

These operators can always be used to compare two values of *numeric types* (such as integers and floats). On both sides of these operators, you can write any expression of a numeric type. These can be constant values, variables, or complete expressions with arithmetic operators (which you’ve seen in Chap. 4).

= Versus == Watch out: in C#, you should write a *double* equals sign (==) if you want to check if two things are equal. This is because the *single* equals sign is already used for assignments. The difference between these two operators is very important:

`x=5;` is an *instruction* that assigns the value 5 to `x`.
`x==5` is a *Boolean expression* that checks whether the value of `x` is equal to 5.

6.2.2 Logical Operators

In logical terms, a condition is also called a *predicate*. You can combine predicates into larger predicates using the so-called **logical operators**: *and*, *or*, and *not*. In C#, these operators have a special notation:

`&&` is the logical *and* operator
`||` is the logical *or* operator
`!` is the logical *not* operator

For instance, if you have two Boolean expressions `conditionA` and `conditionB`, then the expression `conditionA && conditionB` is another Boolean expression. This expression is only true if `conditionA` and `conditionB` are *both* true; otherwise, it is false. The expression `conditionA || conditionB` is true if *at least one* of the two parts is true. Finally, the expression `!conditionA` has exactly the opposite value of `conditionA`, so you can use it to “flip” expressions between **true** and **false**.

Just like with arithmetic operators such as `+` and `*`, you can group these operators into larger expressions. For example, the following code (in a hypothetical game) draws a “You win!” image only if the player has more than 10000 points, *and* the enemy has a life force of 0, *and* the player life force is larger than 0:

```
if (playerPoints > 10000 && enemyLifeForce == 0 && playerLifeForce > 0)
{
    spriteBatch.Draw(winningImage, Vector2.Zero, Color.White);
}
```

And just like with arithmetic, logical operators all have a certain priority. The `!` operator has the highest priority, followed by `&&`, and then followed by `||`. So in the following expression:

`conditionA || conditionB && conditionC`

the expression `conditionB && conditionC` will be considered first, and the outcome of that will be given to the `||` operator. If you want to change this priority order, you can use brackets, just like in mathematical expressions. In the following expression:

`(conditionA || conditionB) && conditionC`

the expression `conditionA || conditionB` will be evaluated first, and the result will be fed to the `&&` operator. The two expressions above are not the same, so watch out for this!

As soon as logical expressions get longer, it becomes harder to keep track of what they mean. In such cases, it may be convenient to store parts of these expressions in helper variables. This is possible because all logical expressions have **bool** as their type, which means that you can store them in *variables* of that type.

6.2.3 Boolean Variables

You've seen how to build Boolean expressions via the comparison operators and logical operators. Because **bool** is a data type (in fact, it's a primitive type just like **int**), you can use it in many ways. For example, you can store the result of a logical expression in a *variable* of type **bool**, or you can pass it as a parameter to a method.

Here's an example of a declaration and an assignment of a Boolean variable:

```
bool test;  
test = x>3 && y<5;
```

In this case, if *x* contains (for example) the value 6 and *y* contains the value 3, then the Boolean expression *x*>3 && *y*<5 will evaluate to **true**. This value will be stored in the variable *test*.

You can also store the Boolean values **true** and **false** directly in a variable, such as in this example:

```
bool isAlive = false;
```

Boolean variables are extremely handy to store the status of different objects in the game. For example, you could use a Boolean variable to store whether or not the player is still alive, or if the player is currently jumping, or if a level is finished, and so on.

Remember: an if-condition (the code between brackets after the keyword **if**) can be any expression of type **bool**. So it can be a comparison of two values, a logical expression, the name of a *variable* that stores a Boolean value, or any combination of these things.

Here's an example that uses the Boolean variable *isAlive* directly as an expression in an **if** instruction:

```
if (isAlive)  
{  
    // do something  
}
```

In this case, if the expression *isAlive* evaluates to **true**, the body of the **if** instruction will be executed. You might think that this code generates a compiler error and that we need to do a comparison of the Boolean variable, like this:

```
if (isAlive == true)  
{  
    // do something  
}
```

However, this extra comparison is redundant. The if-condition has to be a Boolean expression, but the *isAlive* variable is already exactly that. You don't have to explicitly compare that variable to **true**: that will just give you a longer Boolean expression with the exact same outcome. Similarly, the expression *isAlive* == **false** could just be shortened to **!isAlive**, using the "not" operator.

So, you can use the **bool** type to store complex expressions that are either **true** or **false**. As a final test, take a look at the following lines of code:

```
bool a = 12 > 5;
bool b = a && 3+4==8;
bool c = a || b;
if (!c)
    a = false;
```

What are the values of the *a*, *b*, and *c* variables after these instructions have been executed? In the first line, you declare and initialize a Boolean *a*. The value stored in this Boolean is read from the expression `12 > 5`, which evaluates to **true** (because 12 is indeed larger than 5). So, variable *a* will store the value **true**. In the second line, you declare and initialize a new variable *b*, in which you store the result of a more complex expression. The first part of this expression is the variable *a*, which contains the value **true**. The second part of the expression is a comparison `3+4==8`. This comparison is not true (`3 + 4` doesn't equal 8), so this evaluates to **false**. Therefore the logical *and* operation on this line also results in the value **false**. As a result, the variable *b* will store the value **false**.

The third line stores the result of the logical “or” operation on variables *a* and *b* in a third variable *c*. Since *a* contains the value **true**, the outcome of this “or” operation will also be **true**, and that value will be assigned to *c*. Finally, we have an **if** instruction that assigns the value **false** to variable *a*, but only if the expression `!c` evaluates to **true**— in other words, if the variable *c* currently stores the value **false**. In this particular case, *c* stores the value **true**, so the body of the **if** instruction will not be executed. Therefore, once all the instructions are executed, *a* and *c* will contain the value **true**, and *b* will contain the value **false**.

6.3 More on if Instructions

It will occur quite often that you want to write *alternative code* below an **if** instruction. This is the code that should be executed only when the if-condition does *not* hold. For example, let's say you want to set the angle of the cannon barrel to zero whenever the left mouse button is *not* pressed. You could write a second **if** instruction for this, containing exactly the opposite condition:

```
if (mouse.LeftButton == ButtonState.Pressed)
{
    // code that calculates the angle
}
if (mouse.LeftButton != ButtonState.Pressed)
{
    angle = 0;
}
```

6.3.1 The *else* Keyword

However, there is a nicer way of dealing with this: by combining an **if** instruction with an **else** instruction. You use the keyword **else** for this, followed by a block of instructions. This block will be executed if the condition in the **if** instruction is *not* true. Our example then becomes this:

```
if (mouse.LeftButton == ButtonState.Pressed)
{
    double opposite = mouse.Y - barrelPosition.Y;
```

```

    double adjacent = mouse.X - barrelPosition.X;
    angle = (float)Math.Atan2(opposite, adjacent);
}
else
{
    angle = 0;
}

```

This instruction does exactly the same thing as the two **if** instructions before, but using **else** prevents you from having to write two separate **if** conditions, because **else** always refers to “the other case” automatically. You can imagine that this is very useful if you ever want to change the first if-condition: you only have to change it in one place then.

Incorporate this example code into the Update method of the Painter1 game, and then compile and run the program. You’ll see that the angle of the cannon barrel will be reset as soon as you release the left mouse button.



CHECKPOINT: Painter1b

The result can also be found in the Painter1b example program of this chapter.

The **else** instruction is *optional*: you don’t *have* to add an **else** after every **if**. On the other hand, every **else** *must* come immediately after an **if**; otherwise, it’s not clear what you would mean by “the other case.” Furthermore, you can (again) leave out the curly braces of the **else** block if that block only contains one single instruction. To summarize:



Quick Reference: **if** and **else**

Immediately after an **if** instruction, you are allowed (but not obliged) to write an **else** instruction:

```

if (myCondition)
{
    ...
}
else
{
    ...
}

```

If the Boolean expression `myCondition` is **true**, then the code inside the **if** block will be executed and the **else** block will be skipped. Conversely, if `myCondition` is **false**, then the **if** block will be skipped and the **else** block will be executed. Just like with **if**, you can omit the curly braces of the **else** block if that block contains only one instruction.

6.3.2 A Number of Different Alternatives

When there are more than two categories of values, each with their own code that you want to execute, you can use multiple **if** and **else** instructions to separate all these cases. The trick here is that **if** and **else** blocks can contain all sorts of instructions, including **if** and **else** instructions again! That way, you can “nest” these instructions as much as you want.

The idea here is to start with an **if** instruction that checks if your value belongs to the first category. If not, the **else** block will be executed. This block starts with another **if** instruction that checks if your value belongs to the second category. If this test fails, the program goes to another **else** block (inside the main **else** block it was already visiting) and so on.

As a toy example, the following program fragment determines within which age segment a player falls, so that you can draw different player sprites:

```
if (age < 4)
{
    spriteBatch.Draw(babyPlayer, playerPosition, Color.White);
}
else
{
    if (age < 12)
    {
        spriteBatch.Draw(youngPlayer, playerPosition, Color.White);
    }
    else
    {
        if (age < 65)
        {
            spriteBatch.Draw(adultPlayer, playerPosition, Color.White);
        }
        else
        {
            spriteBatch.Draw(oldPlayer, playerPosition, Color.White);
        }
    }
}
```

In this example, every **else** instruction (except the last one) has another **if** instruction inside its block. If the age value is smaller than 4, the `babyPlayer` sprite gets drawn, and the rest of the instructions are ignored (they are inside the **else** block, after all). If the age value is at least 65, the program goes through all the tests (younger than 4? younger than 12? younger than 65?) before it concludes that it has to draw the `oldPlayer` sprite.

Indentation and braces are used in this program to indicate which **else** belongs to which **if**. When there are many different categories, the text of the program becomes less and less readable. Because combining **if** and **else** instructions like this happens so often, C# allows you to write this code in a slightly shorter way: it lets you replace **else { if { ... } }** by **else if { ... }**. The code below does exactly the same as before, but it's more comfortable to look at:

```
if (age < 4)
{
    spriteBatch.Draw(babyPlayer, playerPosition, Color.White);
}
else if (age < 12)
{
    spriteBatch.Draw(youngPlayer, playerPosition, Color.White);
}
else if (age < 65)
{
    spriteBatch.Draw(adultPlayer, playerPosition, Color.White);
}
else
{
    spriteBatch.Draw(oldPlayer, playerPosition, Color.White);
}
```

The main advantage of this layout is that it's much easier to see which cases are handled in which order. (To make the code even shorter, you could remove all the curly braces here, because each block only contains one instruction.)

6.3.3 Handling Mouse Clicks Instead of Presses

As a final example of using the **if** instruction to handle mouse button presses, let's handle a mouse button *click* instead of a mouse button *press*. You'll change the Painter example so that the cannon starts or stops following the mouse pointer at every click. So, the cannon initially stands still until you click the left mouse button for the first time. The cannon then follows the mouse pointer until you click the left mouse button again and so on.

The solution to this problem can be found in the example program Painter1c, but we will guide you through the programming process now. First, create a member variable of type **bool** that will store whether or not the cannon barrel is currently following the mouse pointer:

```
bool calculateAngle;
```

Initialize this variable inside the constructor of your class:

```
calculateAngle = false;
```

And inside the Update method, use the **if** and **else** keywords to compute a new angle only if calculateAngle is currently true:

```
if (calculateAngle)
{
    double opposite = mouse.Y - barrelPosition.Y;
    double adjacent = mouse.X - barrelPosition.X;
    angle = (float)Math.Atan2(opposite, adjacent);
}
else
{
    angle = 0.0f;
}
```

The **else** block means that the cannon barrel will return to its horizontal position whenever the mouse button is not pressed.

What remains is to switch the calculateAngle variable from **false** to **true** (or vice versa) when the user clicks the left mouse button. Unfortunately, the mouse state of MonoGame only stores whether or not a mouse button is currently held down, but it doesn't say in which frame the player has *started* or *stopped* pressing a mouse button. It's a shame that MonoGame doesn't provide this information for you, but don't worry: by using member variables and **if** instructions, you can easily create this yourself!

The idea is to not only use the mouse state of the *current* frame but also of the *previous* frame. After all, if the player has just started pressing a mouse button, the button state should be ButtonState.Pressed in the current frame, whereas it was still ButtonState.Released in the previous frame. Similarly, if the player releases the mouse button, the current button state should be ButtonState.Released, whereas the previous state was ButtonState.Pressed.

We say that the player has *clicked* the mouse button if the following two conditions hold:

1. the player is pressing the mouse button in the current Update method;
2. the player was not yet pressing the mouse button during the previous Update method.

To implement this, you can store the mouse state of the previous frame in a member variable. That way, you can always compare the current mouse state to the state that you saved previously. Let's create two member variables for the current and previous state:

```
MouseState currentState, previousMouseState;
```

In the Update method, you can set the `currentState` variable easily:

```
currentState = Mouse.GetState();
```

Now, how do we get the *previous* mouse state? There is no way of doing this directly by calling a method from the Mouse class. However, you do know that what is now the *current* mouse state will become the *previous* mouse state in the next Update call. So, you can solve this problem by “copying” the current mouse state into the `previousMouseState` variable. If you do this just before you overwrite `currentState` with a new value, you will have two different mouse state objects that you can compare.

In short, start the Update method with the following two lines:

```
previousMouseState = currentState;
currentState = Mouse.GetState();
```

Then add the following instruction, which checks whether or not the user has just started pressing the left mouse button, and then stores the result in a helper variable:

```
bool mouseButtonClicked = currentState.LeftButton == ButtonState.Pressed
&& previousMouseState.LeftButton == ButtonState.Released;
```

Next, use this variable as the condition of an `if` instruction that toggles the `calculateAngle` variable:

```
if (mouseButtonClicked)
    calculateAngle = !calculateAngle;
```

The line `calculateAngle = !calculateAngle;` may look a bit weird, but it correctly switches the value of `calculateAngle` from **false** to **true** (or vice versa). After all, `calculateAngle` is a Boolean member variable, so it is either true or false. The expression `!calculateAngle`, using the logical ! (“not”) operator, is another Boolean expression that has exactly the opposite value of `calculateAngle`. By *assigning* this value to `calculateAngle` again, you give that variable the opposite value of what it used to have. As a result, the value of the `calculateAngle` variable will be toggled every time you execute that instruction.

The rest of the Update method should update the angle based on the `calculateAngle` variable, as explained before. You'll now have to use `currentState` (instead of `mouse`) to get the mouse position, but the rest remains the same.



CHECKPOINT: Painter1c

If you combine everything and then compile and run the program, you should see the desired toggling behavior.

6.4 A Multicolored Cannon

Now that you know about `if` instructions and Boolean expressions, it's time to build the next feature of the Painter game. In this section, you're going to change the Painter1 program so that the player can

change the color of the cannon. By pressing the R, G, or B key on the keyboard, the cannon's color will change to red, green, or blue.

The result of this section can be found in the example program Painter2. But as usual, we encourage you to make these changes yourself, starting with Painter1 from the previous chapter. (The Painter1a, Painter1b, and Painter1c projects from the previous sections were just demonstrations; they're side-tracks from our main quest of writing the Painter game.) Again, you may want to save a copy of your project before you continue.

6.4.1 *Choosing Between Multiple Colors of Sprites*

Our goal is to give the cannon a color (red, green, or blue) and to let the player change this color by pressing different keys (R, G, or B). You can display the cannon's current color by drawing a colored ball in the center of the rotating barrel. We've prepared three sprites for this (*spr_cannon_red*, *spr_cannon_green*, and *spr_cannon_blue*) that you can add to the project right now. To store these sprites in the game, declare three new member variables:

```
Texture2D colorRed, colorGreen, colorBlue;
```

Just like all other Texture2D variables, you can initialize these variables in the LoadContent method:

```
colorRed = Content.Load<Texture2D>("spr_cannon_red");
colorGreen = Content.Load<Texture2D>("spr_cannon_green");
colorBlue = Content.Load<Texture2D>("spr_cannon_blue");
```

And to make sure that the ball's texture will be nicely centered on the cannon, add a member variable for storing the origin of the ball:

```
Vector2 colorOrigin;
```

You can initialize this variable in LoadContent as well:

```
colorOrigin = new Vector2(colorRed.Width, colorRed.Height) / 2;
```

(The three color sprites have exactly the same size, so we can choose any of the three to determine the origin.)

Next, we need to keep track of the cannon's current color. You can do this in many ways, but we've chosen to give the class an extra member variable called *currentColor*:

```
Color currentColor;
```

This variable will always store one of three colors: red, green, or blue. We have to give this variable a value when the game starts. Let's start with a blue cannon, by initializing *currentColor* inside the LoadContent method:

```
currentColor = Color.Blue;
```

Inside the Draw method, we can now use **if** and **else** instructions to determine which sprite we need to draw, based on the color stored in the *currentColor* variable. In the Draw method, first declare a helper variable that will store the sprite to draw:

```
Texture2D currentSprite;
```

Then use **if** and **else** instructions to fill this variable with the correct sprite:

```
if (currentColor == Color.Red)
    currentSprite = colorRed;
else if (currentColor == Color.Green)
    currentSprite = colorGreen;
else
    currentSprite = colorBlue;
```

Do you see how we're using the `==` operator here? Luckily for us, MonoGame has *overloaded* the `==` operator for the `Color` data type, so you can use it to check if two colors are equal. For example, the expression `currentColor == Color.Red` will have the value **true** if `currentColor` stores exactly the same R, G, B, and A components as `Color.Red`. If this expression is **false**, then `currentColor` apparently does not contain the color red, and we continue by checking for green (and then for blue).

After you've chosen the correct sprite, you can finally draw it:

```
spriteBatch.Draw(currentSprite, barrelPosition, null, Color.White, 0f,
    colorOrigin, 1.0f, SpriteEffects.None, 0);
```

Make sure to draw the color sprite *after* the background and the barrel, so that it won't get hidden behind other sprites.

Note that the color sprite is drawn at `barrelPosition`: exactly the same position as the cannon barrel sprite. Combined with `colorOrigin`, this makes sure that the sprite is drawn exactly in the middle of the barrel's base.

If you compile and run the game now, you should see a blue image on the cannon. Next up, let's add keyboard interaction so that the player can *change* this color.

6.4.2 Handling Keyboard Input

In MonoGame, **keyboard input** can be handled very similarly to mouse input. But instead of getting the mouse state, you have to retrieve the *keyboard* state. This can be done as follows:

```
KeyboardState currentKBState = Keyboard.GetState();
```

As you can probably guess by now, `KeyboardState` is a MonoGame class that gives you all kinds of information about the current keyboard state. The class has several methods for checking if the player presses a key on the keyboard. For example, you can check if the *A* key on the keyboard is pressed by calling `currentKBState.IsKeyDown(Keys.A)`. This method returns either **true** or **false**—in other words, it's a Boolean expression! This means that you can immediately use these methods in **if** instructions, too.

By the way, `Keys` is another enumerated type (just like `ButtonState`). It's part of the MonoGame engine, and it defines all possible keyboard keys that a player can press.

But the `IsKeyDown` method can only tell us whether or not a key is currently down; it doesn't tell us whether the user has *started pressing a key* in this frame. To account for this, you have to store the *previous* keyboard state in memory, just like how we stored the previous mouse state earlier in this chapter.

For completeness, let's store the previous and current state for both the keyboard and the mouse. This means you need the following member variables:

```
MouseState currentMouseState, previousMouseState;
KeyboardState currentKeyboardState, previousKeyboardState;
```


In the `Update` method, you first have to update these variables: the previous states should store the previously known mouse and keyboard states, and the current variables should store the states that `MonoGame` currently knows.

```
previousMouseState = currentMouseState;
previousKeyboardState = currentKeyboardState;
currentMouseState = Mouse.GetState();
currentKeyboardState = Keyboard.GetState();
```

With this in place, the following Boolean expression evaluates to **true** when the player has just started pressing the R key:

```
currentKeyboardState.IsKeyDown(Keys.R) && previousKeyboardState.IsKeyUp(Keys.R)
```

Note that this is very similar to what we did with mouse clicks earlier. Because this is a Boolean expression, we can use it in an `if` instruction to change the cannon's color. Add the following code to the `Update` method:

```
if (currentKeyboardState.IsKeyDown(Keys.R) && previousKeyboardState.IsKeyUp(Keys.R))
{
    currentColor = Color.Red;
}
```

If the player has just pressed the R key, this code assigns the color red to the `currentColor` variable. Follow the same procedure for the G and B keys, resulting in the following code:

```
if (currentKeyboardState.IsKeyDown(Keys.R) && previousKeyboardState.IsKeyUp(Keys.R))
{
    currentColor = Color.Red;
}
else if (currentKeyboardState.IsKeyDown(Keys.G) && previousKeyboardState.IsKeyUp(Keys.G))
{
    currentColor = Color.Green;
}
else if (currentKeyboardState.IsKeyDown(Keys.B) && previousKeyboardState.IsKeyUp(Keys.B))
{
    currentColor = Color.Blue;
}
```



CHECKPOINT: Painter2

Compile and run the program now, and see how the program responds to presses of the R, G, and B keys. The color of the cannon will change accordingly.

Again, notice the “magic” of the game loop here: because the code in `Update` and `Draw` gets executed in each frame, the game will always show the result of the player's most recent action. Nice work!

6.5 What You Have Learned

In this chapter, you have learned:

- what an enumerated type is;
- how to use the `if` instruction to execute code only when certain conditions hold;
- how to formulate conditions for these instructions using Boolean values;
- how to use `if` and `else` to deal with different alternatives;
- how to apply this to keyboard and mouse handling in `MonoGame`.

6.6 Exercises

1. *Enumerated Types*

What is an enumerated type, and what are the advantages of using it instead of an integer? Name a few examples of situations in which you could use an enumerated type (other than the examples you've already seen in this chapter).

2. *Strange Usage of **if** and **else***

Given two **int** variables (a and b), both with a meaningful value, what does the following code do? Can you write a shorter version of this program?

```
if (a != 0)
    b = a;
else
    b = 0;
```

3. *Mixed Seasoning*

Let's say we have two **int** variables (day and month) that together represent a valid day of the year. The following code tries to draw a background sprite based on the *season* expressed by this day and month. For simplicity, we only look at spring and summer, so we assume that the value of month lies between 4 and 8 (meaning between April and August). Note: The (astronomical) summer season begins on June 21.

```
if (month < 6)
{
    spriteBatch.Draw(backgroundSpring, Vector2.Zero, Color.White);
}
else if (day < 21)
{
    spriteBatch.Draw(backgroundSpring, Vector2.Zero, Color.White);
}
else
{
    spriteBatch.Draw(backgroundSummer, Vector2.Zero, Color.White);
}
```

- Does this code always draw the correct background sprite? (*Hint*: no!) Find a combination of day and month that gives us the wrong answer.
- Write a variant of this code that *does* work correctly for all dates (between April and August).
- Extend your code to support autumn and winter as well.

4. * *Bouncing Balloons*

In the FlyingSprites example from Chap. 5, a balloon sprite moves upward according to the game time.

- Modify the program so that the balloon has a separate *velocity* vector (another **Vector2** member variable), indicating the balloon's horizontal and vertical movement in pixels per second. Use this velocity to update the balloon's position in the **Update** method. Verify that you can use different vector coordinates to let the balloon move in different directions.
- Modify the program so that the balloon bounces back when it touches the edge of the screen. *Hint*: use an **if** instruction to check for this, and change the *x* or *y* component of the balloon's velocity when the **if**-check succeeds.

Chapter 7

Basic Game Objects



The Painter game is already getting more complex. This chapter will start organizing the Painter code into methods and classes. You will first learn how to write your own *methods*, with or without parameters. Next, you will see how to create your own *classes*, and how these relate to the *objects* in your game. Along the way, you will also learn all about *properties* in C#.

This is one of the most important chapters in the book, because this is where your experience with **object-oriented programming** really starts. We'll cover many of the topics that we promised to get back to. This also is one of the *longest* chapters, because we want to give you a full understanding of these topics.

You'll apply this new knowledge to the Painter game by creating separate classes for the cannon, for input handling, and for the game world. At the end of this chapter, your Painter game will still do the same as before, but the code will be organized in a better way. Organizing complicated programs (such as games) into methods and classes is one of the most important tasks of a programmer. By grouping your instructions in methods and classes, you can more easily recognize which part of your code does what. A second advantage is that you can reuse those methods and classes in multiple places, without having to write the code multiple times.

We'll use the result from Chap. 6 as a basis. If you've followed the book so far, you can continue working with your result from the previous chapter. Alternatively, you can start with the Painter2 example program from that chapter. You can find the results for this chapter in example programs again, but (as usual) we will work towards them step by step.

This chapter will assume that your game's main class is called Painter. If it's still called Game1 (or something else), it would be useful to change the name to Painter now.

7.1 Grouping Instructions into Methods

Recall that a **method** is a group of instructions. If you *call* a method elsewhere in your code, the instructions inside that method will be executed. So far, you've used methods that already existed in MonoGame. In this section, you will start writing (and calling) your *own* methods.

Methods are particularly useful in the following two scenarios:

- You have multiple instructions that logically fit together, because they're related to a specific task (such as input handling or drawing sprites). By turning this group of instructions into a method, you'll nicely separate one concept from the rest of your code.

- You have a group of instructions that you need in multiple places in your program. By turning this group into a method, you can simply call this method when you need it, instead of having to copy and paste the instructions everywhere.

Let's start by creating a method for the first reason.

7.1.1 A Method for Input Handling

Take a look at the `Update` method of `Painter`: currently, all of its instructions are related to input handling. It's a good idea to create a separate method for this task. That way, if you ever want to look up how the game responds to player input or if you ever want to *change* the input handling in some way, you immediately know where to look.

So, create a new method by writing the following code below (or above) the `Update` method:

```
public void HandleInput()
{
}
```

Let's look at all elements of the **method header** here:

- The word `HandleInput` is a name we've chosen ourselves. You could use any name you want (except for the names that are already taken by other methods). In general, it's a good idea to give a method a useful name that indicates what the method does. In this case, `HandleInput` makes sense because we're going to use the method for input handling.
- The two brackets after `HandleInput` indicate that this method *has no parameters*. Later in this section, we'll discuss methods that *do* use parameters, but we don't need them just yet.
- The keyword **`void`** indicates the **return type** of this method. Specifically, the word **`void`** means that this method *does not return a result*. In other words, if you call this method, it will just execute some instructions and then finish, without returning an object that you can (for example) store into a variable. Later in this section, we'll talk about methods that *do* return a result.
- The keyword **`public`** is a so-called **access modifier**, which says how accessible this method is throughout your program. Specifically, the word **`public`** roughly means that you can call this method everywhere in your program. There are other access modifiers that are more restrictive (such as **`private`** and **`protected`**), but we'll get to that later in this chapter. For now, **`public`** is a fine choice, and we can always still change it if we want.

Next, take *all* instructions from the `Update` method, and move them to the body of the `HandleInput` method. This means that `HandleInput` now contains the instructions related to input handling. The `Update` method now has an empty body.

But remember: a method doesn't do anything until you *call* it! The `Update` method is called automatically, thanks to `MonoGame`'s game loop, but if you create a method yourself, you're also responsible for calling it yourself. In other words, you now have to call the `HandleInput` method inside the `Update` method:

```
protected override void Update(GameTime gameTime)
{
    HandleInput();
}
```

That way, the instructions in `HandleInput` will be executed in every frame of the game loop (just like in the previous version of the program).

The instruction `HandleInput();` is a **method call** just like the ones you’ve seen before, except that you’re now calling a method you’ve created yourself. A method call always consists of the name of the method (to tell the compiler which method you mean) and a pair of brackets with (possibly) parameters in-between. To turn the method call into an *instruction*, we have to write a semicolon behind it, too.

Compile and run the program now. If you’ve made the correct changes, the game should still do the same thing as before.

Currently, the `Update` method doesn’t do anything else yet, but you can imagine that it’ll do much more in more complicated games: handle collisions between game objects, calculate speeds and positions of objects, update the current score, and so on. It will make sense to create separate methods for each of these tasks, so that the `Update` method itself won’t contain hundreds of instructions.

In C#, you can add any number of methods to your class. You can also write the methods in any order: the behavior of your program only depends on when you *call* which method, but it doesn’t depend on the order in which you promise that the methods *exist*.

Whenever you add a method to a class, think about *why* you do it. For example, it would probably not be very useful to add a method that loads the background sprite: that method would contain only one instruction, and you can just as well write that instruction directly. Furthermore, this method would solve only one very specific problem (loading the background sprite) and you would call it only once.

As you turn into a more experienced programmer, you will get a better feeling of when to use methods (and classes). This is something you have to learn by simply trying it many times. That’s a good excuse for creating a lot of games!

7.1.2 *Methods with Parameters*

When explaining the header of the `HandleInput` method, we said that this method has no parameters, does not return a result, and is accessible everywhere. These three things are choices that we’ve made ourselves. As a programmer, you make these choices for each new method that you write.

The `HandleInput` method didn’t have parameters because we always want it to do the same thing. An example of a method that *does* have parameters is the `Clear` method of `GraphicsDevice`. This method has a parameter that indicates the background color that you want the method to draw. You’ve already played with this parameter a lot, to change the background color of `DiscoWorld`.¹

To add **parameters** to your method, you write these parameters between the brackets of your method header. You can see an example of this in the header of `Update`:

```
protected override void Update(GameTime gameTime)
```

Here, `GameTime` is the *type* of the method’s parameter. A parameter can have any data type (such as an `int`, a `Color`, or any class type that you’ve created yourself). The second word, `gameTime`, is a name chosen by the programmer. It’s common (but not mandatory) to start the name of a method parameter with a lowercase letter.

¹That already feels like *decades* ago, doesn’t it?

You can give a method multiple parameters by separating them with commas. For example, here's a method that takes three integers as parameters and uses these integers to draw a background color:

```
public void DrawBackgroundColor(int r, int g, int b)
{
    GraphicsDevice.Clear(new Color(r,g,b));
}
```

Each time you call the `DrawBackgroundColor(...)` method, you need to fill in three integers at the position of "...", separated by commas. These values will then be used inside the method.

Default Parameter Values — By the way, it's possible to give parameters a *default value* in your method header. This allows programmers to leave out those parameters, and then they'll automatically be filled in. For example, you could change the header of `DrawBackgroundColor` as follows:

```
public void DrawBackgroundColor(int r, int g, int b=255)
```

If you now call `DrawBackgroundColor(...)`, you can still give it three integers to work with, but you're also allowed to give it only *two* and leave out the third one. This third parameter will then automatically receive a value of 255, in this case. You won't use default parameter values yourself soon, but it'll happen later in this book.

7.1.3 The Scope of Method Parameters

The `DrawBackgroundColor` method above is not very useful because it only contains a single instruction, but it does show something interesting. Inside a method, you can use the method's parameters just like variables. The **scope** of these parameters is restricted to that method alone. So, inside the `DrawBackgroundColor` method, you can write `r`, `g`, or `b` and the compiler will know that you mean the method's parameters. But outside the method, the compiler will *not* know this, and `r`, `g`, and `b` will be undefined things.

This makes sense if you realize that you can call a method multiple times, with different parameters each time. Every time you call a method, the parameters of that method are filled in with the values you provide. For instance, let's say you've written the instruction `DrawBackgroundColor(0,0,0);` somewhere. Whenever your program reaches this instruction, it "jumps" into the `DrawBackgroundColor` method, and it fills the `r`, `g`, and `b` parameters with the value 0. But if the program reaches an instruction `DrawBackgroundColor(255,255,255);`, it will jump into the method and fill these parameters with the value 255.

In other words, you could see method parameters as special types of variables that only exist inside that method. These variables can have a different value each time, depending on how the method is called. This explains why the compiler cannot possibly know what you mean if you try to refer to these parameters outside the method.

By the way: it can happen that your class has a member variable with the same name as a method parameter. The member variable and method parameter can even have different types! This is allowed, but you need to watch out a bit if you ever want to use both versions at the same time.

For instance, let's say your game class has a member variable `Texture2D r` that stores a sprite. If you write `r` in any method of your class, the compiler will assume that you mean the `Texture2D r` member variable. But if you write `r` inside the `DrawBackgroundColor` method, the compiler will assume that you

mean the **int** *r* method parameter! Thus, the compiler always chooses the most specific version that it knows about.

But what if you explicitly want to use the member variable `Texture2D r` inside the `DrawBackgroundColor` method? In that case, the trick is to write **this.r** instead of *r*. The **this** keyword tells the compiler that you're looking for a member variable of the game class. We'll talk more about the **this** keyword later in this chapter, because it has a lot to do with classes and instances.

You can often avoid these kinds of situations by giving variables and parameters more meaningful names, such as `redSprite` or `redColorComponent`. In larger programs, though, name clashes can still occur, so it's good to have heard about this.²

7.1.4 Methods with a Result

Next, let's look at methods that return a result. You've already seen such methods before. For example, the `GetState` method from the `Keyboard` class returns an object of type `KeyboardState`. You've stored this result in a variable as follows:

```
currentKeyboardState = Keyboard.GetState();
```

Here, the method call `Keyboard.GetState()` executes the instructions inside that method (just like any other method call). The *result* of that method call is stored inside the `currentKeyboardState` variable.

The `HandleInput` method that you just made does *not* return a result. Of course, that method does have an effect of some sort: it interprets the input from the player and changes some member variables accordingly. However, when talking about a *method with a result*, we mean that the method should return a concrete expression that you can use further in your program. This expression is also called the **return value** of a method. The keyword **void** indicated that `HandleInput` does not have a return value.

So how do you write a method that *does* have a return value? First, you should replace the keyword **void** by the *type* of object that your method returns. (This can be any data type again.) Second, inside the method itself, you should “build up” this result and then return it when you're done.

An intuitive example of a method with a result is a *mathematical function*. In mathematics, a function is something that takes an input value and returns an output value based on that input. For example, the function $f(x) = x^2$ takes as parameter an x value, and it returns the square of x (which is $x \cdot x$) as its result. You could write this mathematical function in C# as follows:

```
public float Square(float x)
{
    return x*x;
}
```

A couple of new things are going on here. The header of this method shows that the method has one parameter of type **float** (as indicated by **float x** between brackets). The header also says that the method has a *return type* of **float** (as indicated by the first **float** keyword). The return type (so: the type of the return value) is always written in front of the method's name. (And remember: the keyword **void** is a special “return type” that indicates that a method doesn't return anything at all.)

The return type **float** means that this method will give you an *expression* of type **float** when you call it. This means that you can (for example) store this result in a variable, as follows:

```
float f = Square(10.0f);
```

²... and to have heard about **this**!

When the program reaches this instruction, it will jump into the `Square` method, filling the parameter `x` with the value `10.0f`. The method will calculate the value of the expression `x*x`, which is another **float** expression that turns out to be `100.0f` in this case. Next, the **return** keyword says that the program should “jump out” of the `Square` method again and that it should return this expression of `100.0f` so that you can store it in a variable. In this case, we store it in a variable **float** `f`.

Thus, the keyword **return** says that the method should stop and that it should return the expression written immediately after that keyword. Of course, the *type* of the expression that you return needs to match the return type of your method. Otherwise, the compiler will complain, because you’re trying to return something that doesn’t match what you *promised* to return. We can’t let the `Square` method suddenly return a color, for example.

And again, calling the `Square` method with different parameters will give you different **float** expressions. The compiler doesn’t have to know in advance what the exact values will be. All it knows is that these values will have **float** as their type.

In summary, the following rules hold:

- If a method returns something, then *calling* that method gives you an expression of the method’s return type.
- A method will return the expression that’s written immediately after the **return** keyword. This expression must have the same type as the return type that you’ve promised.
- The expression returned by a method call can be different each time, for example, if it depends on the method’s parameters.
- When compiling the game, the compiler only needs to know the *type* of expression that a method returns. It doesn’t care about the actual value yet.

7.1.5 More on the **return** Keyword

We said that the **return** keyword will terminate a method and return the expression immediately to the right of that keyword. This means that any instructions *after* the return instruction will not be executed. For example, consider the following method:

```
public int SomeMethod()
{
    return 12;
    int tmp = 45;
}
```

In this example, the second instruction (`int tmp = 45;`) will never be executed, because the instruction before it will end the method. This may seem pretty useless (why would you even write this second instruction then?), but you can actually use it to your advantage sometimes. Take a look at the following example of a method that computes the square root of a number:

```
public float SquareRoot(float f)
{
    if (f < 0.0f)
    {
        return 0.0f;
    }
    else
    {
        float result = ... // calculate the square root of f somehow
        return result;
    }
}
```


You cannot calculate the square root of a negative number. Therefore, this example uses an **if** instruction to check if the method has invalid input. Then, in the **else** block, you can be sure that the input parameter has a valid value.

If the if-condition is true, then this method returns a value of 0.0f (because it still has to return *something*). In the other case, the method will actually calculate a square root and then return it.

This example also shows that you can have multiple **return** instructions inside the same method. Especially when combined with **if** statements, you can have multiple “branches” inside your method where each branch returns something else.

Something interesting is going on, though. Whenever the **if** block gets executed, you know for sure that the method will stop *before* it leaves that **if** block! In other words, if the program ever reaches a point *below* the **if** block, you know for sure that the if-condition was false. This means that you don’t need the **else** instruction anymore! Thus, the following program does exactly the same:

```
public float SquareRoot(float f)
{
    if (f < 0.0f)
    {
        return 0.0f;
    }
    float result = ... // calculate the square root of f somehow
    return result;
}
```

Do you see why this is correct?

7.1.6 More on Return Values

The return value of a method doesn’t *have* to get stored in a variable. In the end, a method call (to a method with a return value) is an expression just like any other expression. So besides storing it in a variable, you can also use it immediately in another instruction. You’ve already seen an example of this in the Painter game:

```
if (currentKeyboardState.IsKeyDown(Keys.R) && previousKeyboardState.IsKeyUp(Keys.R))
{
    currentColor = colorRed;
}
```

Here, the `IsKeyDown` and `IsKeyUp` methods both return a value of type **bool**, so the method call `currentKeyboardState.IsKeyDown(Keys.R)` is an expression of type **bool**. This means you can use it in an **if** statement, just like any other Boolean expression. We didn’t have to store the results of these two method calls in separate variables first.

By the way, even if a method has a return value, it’s not mandatory to do anything with the result. For example, the following instruction is also valid code:

```
Square(10.0f);
```

But this line of code is not very useful: it calculates the square of 10, but it doesn’t store the result in a variable, and it doesn’t pass the result to another method. In this case, the result was probably the whole reason for calling the method! So this instruction is probably a mistake.

Sometimes, though, a method can change something in your class (just like `HandleInput`) *and* return a result. For instance, imagine that we’d extend the `HandleInput` method so that it returns a **bool** value,

indicating whether or not the method was successfully executed. You can then use this return value as follows:

```
bool success = HandleInput();
if (!success)
{
    // do something else
}
```

You can also choose to ignore the return value, by simply calling the method and not using the result:

```
HandleInput();
```

Ignoring the return value here is more useful than in the `Square(10.0f)` example, because the `HandleInput()` method call actually has other effects as well. In summary, whenever you call a method with a return value, it's likely that you'll want to use the result in some way, but you don't *have* to use it.

Finally, remember that a method with the return type **void** doesn't return anything. As a result, you cannot try to store the method's result in a variable either. The following lines will generate a compiler error:

```
float myFloat = DrawBackgroundColor(0,0,0);
int myInt = DrawBackgroundColor(0,0,0);
Texture2D mySprite = DrawBackgroundColor(0,0,0);
```

because `DrawBackgroundColor` doesn't return anything.

But in a **void** method, you can still use the **return** instruction to leave the method earlier. To do that, simply use the keyword **return** without a value behind it. This means that you'll leave the method without returning anything. Here's an example of a `Jump` method that prevents a character from jumping if the character is not alive:

```
public void Jump()
{
    if (!isAlive)
    {
        return;
    }
    // code that makes the character jump.
}
```

You could also write the same method *without* a **return** instruction, by flipping the if-condition:

```
public void Jump()
{
    if (isAlive)
    {
        // code that makes the character jump.
    }
}
```

Both options are valid and give the same result, so you can choose which version you like best. At least it's worth knowing that you're allowed to use the keyword **return** inside a **void** method.

Let's summarize what you know about methods up until now:



Quick Reference: The Ingredients of a Method

A method *header* consists of the following ingredients (in this order):

- An *access modifier*, such as **public** or **private**, indicating where you're allowed to use this method.
- A *return type*, which is the type of expression that your method returns. If your method does not return anything, you use the keyword **void** here.
- A name, chosen by yourself.
- A pair of brackets, containing zero or more *parameters* separated by commas. Each parameter has a type and a name. Inside the method, you can use these parameters as if they are "regular" variables.

A method *body* can contain any number of instructions. If the method does not have **void** as its return type, the body must include the keyword **return** followed by an expression to return. This expression must have the same type as the method's return type.

When the program reaches a **return** instruction, it will not execute the rest of the method. (In **void** methods, you can also use the keyword **return** to stop early.)

7.1.7 Expressions, Instructions, and Method Calls

In Chap. 4, we briefly said that the term "method call" can refer to an instruction *or* an expression. Now that you completely understand what methods are, we can tell you the full story.

Let's say you have a method `MyMethod` with return type **int** (and with zero parameters). The following two pieces of code are *instructions*:

```
int x = MyMethod();  
MyMethod();
```

The first line calls `MyMethod` and stores the result in a variable **int** `x`. The second line also calls `MyMethod`, but it does nothing with the result of that method. This is also valid C#: remember, you don't *have* to use the result of a method. (For methods with the return type **void**, you don't even have a choice!)

By contrast, the following piece of code is an *expression* of type **int**:

```
MyMethod()
```

This piece of code is not an instruction (because it doesn't contain a semicolon). It has to be placed inside a larger instruction (such as in the previous two lines) before its valid C#.

Without context, `MyMethod()` is just an integer expression, just like `42` or `1 + 1`. This means you can use it in larger expressions as well. For example, the following expression:

```
MyMethod() + MyMethod() + 42
```

calls the method twice and calculates the sum of their return values and 42. And the following instruction:

```
int x = MyMethod() + MyMethod() + 42;
```

stores that result in a variable. Each time the program encounters the code fragment `MyMethod()`, it jumps into that method and executes the instructions inside it. But despite that, `MyMethod()` on its own is not an instruction yet. It always has to be part of a bigger instruction.

When people say “method call,” they often mean an instruction such as `MyMethod()`; and not an expression such as `MyMethod()`. That’s how we’ve mostly been using the term “method call” so far in this book, too. But if you want to be really precise, you actually have to say the opposite:

- `MyMethod()` is a **method call**, and it is also an *expression*.
- `MyMethod()`; is an *instruction* that *contains* a method call.

In this book, we will use the term “method call” for both of these things. We admit that this is a bit confusing, but it looks like everyone else is doing it as well.³ We’ve made this choice on purpose: we hope that it helps avoid confusion if you ever look for programming advice online.

Knowing this, we should update our Quick Reference box about expressions:



Quick Reference: Expressions and Operators (2)

An *expression* is a piece of code that has a value of a certain type (such as `int`). An expression can be one of the following things:

- a constant value (such as 12);
- a variable;
- a *method call* (excluding the possible semicolon behind it);
- an operator (such as + or −) with two expressions around it;
- an expression between parentheses (to overrule the standard order of operators).

Expressions can actually be a few more other things, too, but that would go too far for this book. Let’s leave this topic for what it is.

7.2 Organizing a Game into Classes

Now that you know how to organize your instructions by grouping them into methods, it’s time to go a step further by organizing your game into *classes*. The Painter program now contains only one class, which already has quite a lot of member variables:

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;
Texture2D background, cannonBarrel;
Vector2 barrelPosition, barrelOrigin;
```

³General life advice, though: “everyone else is doing it” is not always a good reason to do something.

```
float angle;
Texture2D colorRed, colorGreen, colorBlue;
Color currentColor;
Vector2 colorOrigin;
MouseState currentMouseState, previousMouseState;
KeyboardState currentKeyboardState, previousKeyboardState;
```

Some of these member variables are used for running the game in general (such as `spriteBatch` and `graphics`), others are for representing the colored cannon, and there are some variables for handling player input. You can imagine that this list will become even longer when you'll add the flying ball and the falling paint cans to this game. And then *Painter* is just the first game of this book!

Once you start building more complicated games, having a huge list of member variables will result in code that is impossible to manage. Grouping your code into methods doesn't improve this, so you need a different type of organization.

The solution is to divide the code into multiple classes, instead of the single class we have now. Similarly to how each *method* in our program does a particular type of task, you can let each *class* of the program be responsible for *representing* a particular part of the game. You'll also distribute the member variables over these classes, so that each class has the variables that are relevant for that class only.

This section explains the relation between classes, instances, variables, and methods. We've hinted at this relation a few times before, but you're now ready to see the complete picture. In later sections, you'll use this to add your own classes to the *Painter* game.

7.2.1 Classes, Instances, and Objects

We've said a few times before that a class is a blueprint for objects. For example, if you have written a class called `Character`, then that doesn't mean yet that there are actual characters in the game. For that, you need to create an **instance** of the class, which will be an **object** that has `Character` as its data type. You create such an instance by calling the **constructor** method of the `Character` class, using the keyword **new**:

```
Character myCharacter = new Character();
```

From that point on, `myCharacter` is an object that you can use.

A class describes both *data* and *behavior*. The data is described by *member variables*, and the behavior is described by *methods*.

7.2.2 Instances and Member Variables

If a class has member variables, then each instance of that class can fill in these member variables with its own values. For example, you've already seen that the `Color` struct contains four numbers (the R, G, B, and A components) and that you can create multiple instances of `Color`, each with different values for these numbers.

As another example, the `Character` class could have a member variable `Vector2 position` that stores the character's current position in the game. Each specific `Character` instance that you create (such as `myCharacter`) will then store its own position. And the character instances can change their positions independently, for instance, if one character jumps while the others don't.

If you make the position member variable publicly accessible (we'll see later how you can do that), you can use this member variable in other classes of your program. But you'll always have to specify *which* character's position you want to use, because all characters can have different positions. The following expression would give you the position of the `myCharacter` instance:

```
myCharacter.position
```

By writing `myCharacter.` at the beginning of this expression, you indicate that you want the position of the `myCharacter` instance. `myCharacter.position` is an expression of type `Vector2`, just like `Vector2.Zero` and **new** `Vector2(100,250)`. So you can store this expression in another variable, or pass it as a parameter to a method, and so on.



Quick Reference: Instances and Member Variables

Each instance of a class can have different values for its member variables. In other parts of your program, you can use these member variables by writing the name of the instance (followed by a dot) in front of the variable name.

For example, let's say that a class `MyClass` has a (public) member variable:

```
public int myInt;
```

Elsewhere in the program, if `myInstance` refers to an instance of `MyClass` (i.e., an actual object of type `MyClass`), then the following expression:

```
myInstance.myInt
```

will give you the value of the `myInt` variable of that specific instance.

7.2.3 Instances and Method Calls

In the same hypothetical game with `Character` objects, if you want to let your `myCharacter` object *do* something in the game, you can call the (public) methods of the `Character` class. For instance, if the `Character` class has a method with the header **public void** `Jump()`, then the following line of code will make your character jump:

```
myCharacter.Jump();
```

Again, by writing `myCharacter.` at the beginning of your method call, you indicate that you want to call the `Jump` method for this specific character instance.

As usual, if this method would require parameters, you would have to fill these parameters with specific values in your method call. Let's say your jumping method would have the header **public void** `Jump(float strength)`, where the `strength` parameter indicates how strongly the character will jump upward. When calling this method, you'll then have to fill in an expression of type **float** between the brackets:

```
myCharacter.Jump(10.0f); // like this...
myCharacter.Jump(42.0f * 3.5f); // or this...
// etc.
```

And as usual, the compiler will not care about the specific value that you supply, as long as it has the correct type.



Quick Reference: Instances and Method Calls

A method affects a specific instance of a class. In other parts of your program, you can call these methods by writing the name of the instance (followed by a dot) in front of the method name.

For example, let's say that a class `MyClass` has a (public) method:

```
public void DoSomething() { ... }
```

Elsewhere in the program, if `myInstance` refers to an instance of `MyClass` (i.e., an actual object of type `MyClass`), then the following expression:

```
myInstance.DoSomething()
```

will execute the `DoSomething` method for that specific instance.

7.2.4 Summary: How All Concepts Are Related

Figure 7.1 gives a summary of the relation between instructions, variables, methods, objects, and classes. On one hand, instructions are grouped in methods, and methods are grouped in classes. On the other hand, (member) variables are grouped in objects, and objects are instances of classes (so they have a class as their type).

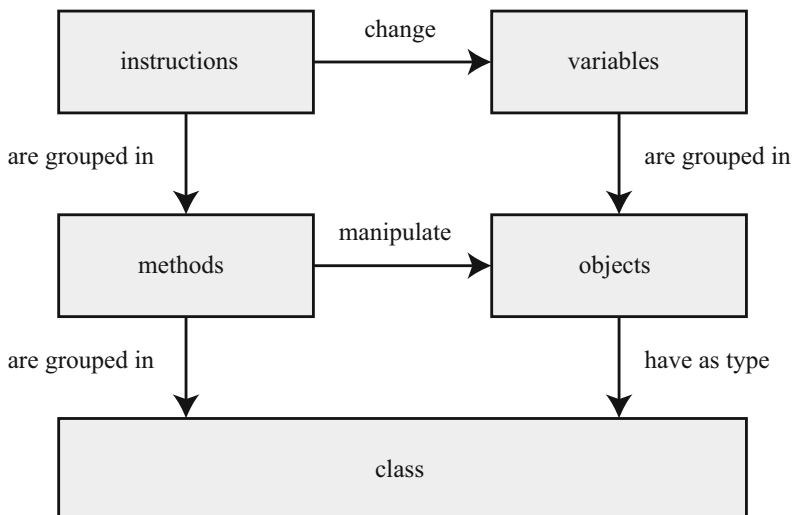


Fig. 7.1 The relation between the core elements of an object-oriented program

Instructions can *change* the variables in an object. And because you execute these instructions by calling the methods in which they're grouped, you could say that methods *manipulate* objects.

7.2.5 Access Modifiers

Let's talk a bit about the **public** keyword that you've seen a couple of times already. This is a so-called **access modifier** that determines which parts of your code can use a certain method or member variable.

There are four different access modifiers in C#. The ones you'll see the most are **public** and **private**. If you write the keyword **public** in front of a method header, you'll be able to call that method everywhere in your program. For instance, let's say (again) that we have a `Character` class with a method with the following header:

```
public void Jump()
```

In any other class, if `myCharacter` refers to an *instance* of a `Character`, then you can call this method as follows:

```
myCharacter.Jump();
```

Likewise, you can make a *member variable* public by adding the keyword **public** to the variable declaration:

```
public Vector position;
```

This allows you to write `myCharacter.position` elsewhere in your code, because the `position` member variable is now accessible from other classes.

If you replace the word **public** by the word **private**, your method or member variable can only be used *inside the class itself*. This way, expressions like `myCharacter.Jump()` and `myCharacter.position` will not be allowed anymore, unless that code is part of the same `Character` class. So, the **private** access modifier is useful for everything that you want to do only inside the `Character` class itself.

Actually, **private** is the default access modifier for all methods and member variables of a class. So if you don't write any access modifier, the compiler will assume that you mean **private**. Later in this chapter, we'll use this to our advantage to make the `Cannon` class (one of the classes you'll write yourself) protective of its own data.

There are two other access modifiers. The keyword **internal** is almost the same as the keyword **public**, but it makes your method or member variable available only inside the same "assembly." In Visual Studio, each project that you create will have its own assembly, so the word **internal** means that you can use a method or member variable within that one project, but not in *other* projects that may be linked to that project. Don't worry about this for now, though: we won't divide our games into multiple projects until Chap. 21.

The final option is called **protected**. This keyword means that you can use a method or member variable only inside that class, but also inside any *subclass* of that class. For example, remember that the `Painter` class is a subclass of `MonoGame's Game` class. Methods like `Update` and `Draw` are marked as **protected**, so that you can use them in your `Painter` class as well. But don't worry about this too much yet: you'll learn all about subclasses in Chap. 10, and we'll get back to this keyword when the time is right.



Quick Reference: Access Modifiers

Within a class, you can use *access modifiers* to define the accessibility of a member variable or a method towards other classes. You write the access modifier in front of the variable declaration or the method header.

- If you use the word **public**, you can refer to the variable or method *everywhere* in your program.
- If you use the word **internal**, you can refer to the variable or method everywhere inside the same *project*.
- If you use the word **protected**, you can refer to the variable or method inside the class itself and in all *subclasses* (the more specific versions of your class).
- If you use the word **private**, you can refer to the variable or method only inside the class itself.

By default, the member variables and methods of a class are *private*. It's good practice to make your classes as protective as possible.

Note: In all cases, you'll still need an *instance* of a class before you can access its member variables or methods. An access modifier does not change that fact; it only changes *from where* you can access a member variable or method.

7.2.6 *this*: The Object That Is Being Manipulated

You've seen that a variable name is also an expression. If a variable `myInstance` stores an instance of a class `MyClass`, then `myInstance` is also an expression that has `MyClass` as its type. This means that you can pass the expression `myInstance` as a method parameter, or use it in more complicated expressions, and so on.

You've also seen variable names being used in another way. To use a method or member variable of a particular class instance, you need a reference to that instance, for example, in the form of a variable. You can then write the name of that variable (followed by a dot) in front of the method or member variable name. For example, `myInstance.DoSomething()` calls the `DoSomething` method of `MyClass`, applied to the specific instance `myInstance`.

But sometimes, you'll be in a situation where you want a class instance to refer to *itself*. That's where a special keyword comes in: **this**. In C#, **this** refers to the object that is currently being manipulated. Or stated differently: it refers to the object that is executing the current code. The concept of **this** may be a bit confusing at first, but you'll come across it more often throughout this book, so let's look at what it does.

For example, imagine you have a `Character` class and a method with the following header:

```
bool CheckCollision(Character other)
```

This method should check if a particular character (the one that receives the method call) has a collision with another character (the one stored in the parameter `other`). In this method, you'd like to write the special case that a character can never collide with itself. To do this, you'd like to check if the character stored in the parameter `other` is the exact same character as the one that has received the method call. How should you do that?

The answer is to use the keyword **this**. Inside the `CheckCollision` method, **this** is an expression of type `Character`, and it refers to the character that has received the current method call. So, the following (Boolean) expression checks if the receiving character is the same as the other character:

```
this == other
```

And you can use this in an **if** instruction to skip collisions of a character with itself:

```
if (this == other)
    return false;
```

In the end, the **this** keyword is just another expression, and its type depends on what class you're in. You can use the expression in many ways, just like how you can use other expressions. For example, you can pass it as a method parameter. You've seen an example of this already, in all `MonoGame` projects so far:

```
graphics = new GraphicsDeviceManager(this);
```

In this case, **this** referred to the instance of the game that we're currently in (and its type was `Painter`, or `Balloon`, or `DiscoWorld`). In `MonoGame`, the constructor of `GraphicsDeviceManager` requires an expression of type `Game` as its parameter. And because a game class (such as `Painter`) is a special version of `Game`, the keyword **this** is exactly such an expression.

You can also write "**this**." in front of a member variable or method name, to emphasize that you're referring to the current object. For example, if a `Character` instance decides that it needs to jump, it can call `this.Jump()`; to ensure that the `Jump` method will be executed for the same `Character` object that's asking for it. This is not *necessary*, though: you can also simply write `Jump()`; and the compiler will already know what you mean. You're still free to use "**this**." if it makes the code clearer for yourself.

However, the prefix "**this**." is *required* if the compiler would otherwise not know what you mean. For example, if the `Character` class contains a member variable `int other` (for whatever reason), then this may give a conflict with the parameter `Character other` of the `CheckCollision` method. Inside that method, if you write the expression `other`, then this expression will refer to the `Character` parameter of the method. This is because method parameters always have priority over member variables. If you want to refer to the member variable `int other` instead, you can write `this.other` to make that wish explicit.



Quick Reference: The Keyword **this**

Inside a method (or a property), the keyword **this** refers to the instance that is currently being manipulated. You can use the keyword in many ways, for example, to:

- let an object pass itself as a parameter to a method;
- let an object compare itself to another object;
- make clear that you're referring to a member variable of the current object (via the prefix "**this**."). This is useful if a method parameter has exactly the same name as that member variable.

Well, this was a bit of a sidetrack, and we can imagine that our discussion of **this** has been somewhat intimidating. But this chapter is really the most logical place to discuss the keyword, because it has everything to do with instances and methods. You won't need it in the rest of the chapter, though, so you have enough time to let it sink in. We'll remind you as soon as the keyword **this** shows its face again.

7.3 Putting It to Practice: Adding a Cannon Class to Painter

It's time to start adding more classes to the Painter game. It makes sense to structure the game world into different *objects*, where each object corresponds to an object in the game. And it's useful to create a class for each *kind* of object that can exist in the game world.

Which objects are present in the Painter game? Well, there are three paint cans that fall down, a ball, and a cannon. As a first step, let's add a class `Cannon` that will represent the cannon object in the game. This class will have only one instance (because there's only one cannon in the game), but it's still a good idea to group its behavior into a class.

7.3.1 Adding a Class File to the Project

To create a new class, you could go to the bottom of your current `.cs` file and start a new class block. A single file can contain as many classes as you want. However, it's common practice among programmers to create a separate `.cs` file for each class and to give each file the same name as the class it contains. That way, if you ever want to find a particular class in your project, it will be much easier to find.

Visual Studio has nice functionality for creating a new class in a new file. Right-click on your Visual Studio project and select `Add → Class`. You will see the window depicted in Fig. 7.2.

Select the "Class" template from the big list that you see. At the bottom of the screen, fill in an appropriate name for the class. In this case, let's call our file `Cannon.cs`. After you click on `Add`, a new

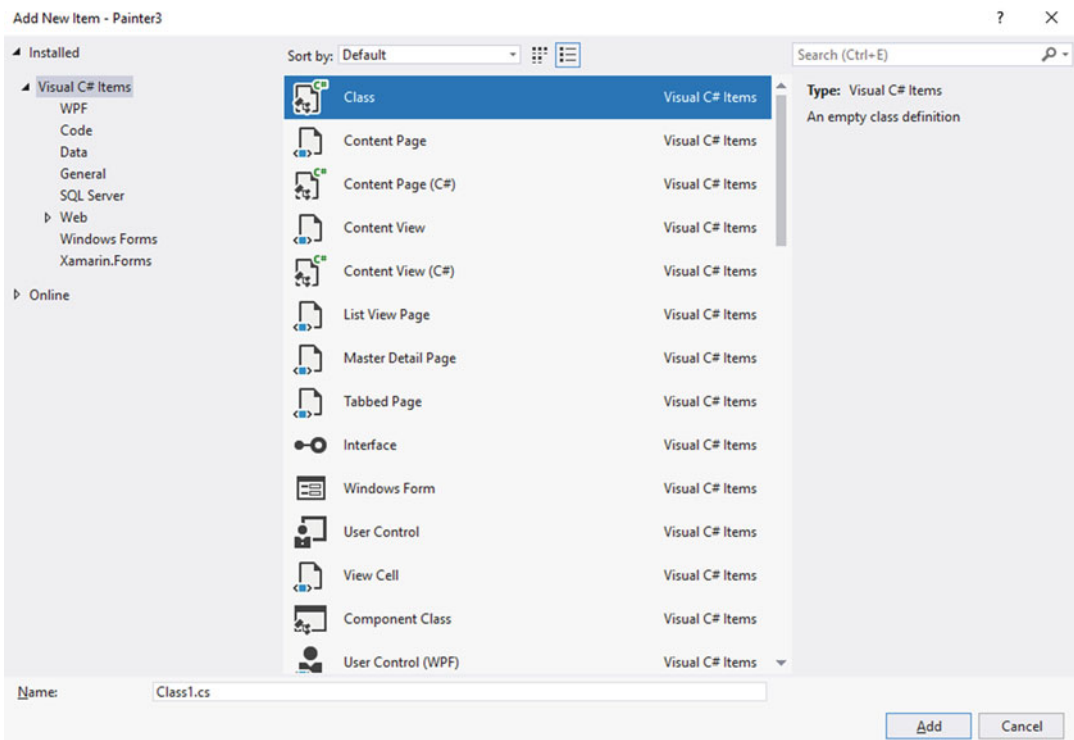


Fig. 7.2 Adding a new class to a project

file called *Cannon.cs* will be added to your project. Double-click on that file to open it. The contents of the file will look something like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Painter
{
    class Cannon
    {
    }
}
```

As you can see, Visual Studio has already created an empty class with the name *Cannon* for you (based on the filename that you've chosen). It has also put this class into a namespace called *Painter* (the name of the project). We will not use namespaces in this example, so you can remove that again.

Finally, Visual Studio has already added a couple of **using** instructions that are often used in Windows applications. You will only need the first one (**using System;**), so you can remove the others if you want.

The simplified version of your file will look like this:

```
using System;

class Cannon
{
}
```

7.3.2 Creating an Instance of *Cannon* in the Game

Currently, the *Cannon* class is completely empty. But even if a class does not yet contain any member variables or methods, you can already create instances of it, and you can already create variables of that type. Add the following member variable declaration to the *Painter* class, just below the large list of existing variables:

```
Cannon cannon;
```

And add the following line at the end of the *LoadContent* method:

```
cannon = new Cannon();
```

The expression **new Cannon()** calls the *constructor* of the *Cannon* class, which results in a new object of type *Cannon* that you store in a variable.

If you remember that constructors are special kinds of methods, you may think that this code is invalid, because we haven't written a constructor for the *Cannon* class yet! However, in C#, a constructor with zero parameters always exists by default for each class. This default constructor simply creates an empty object (of the correct type) without doing anything extra. If you ever want to add your *own* code to the constructor, then you do need to write it yourself—but more about that later.

You have now created an **instance** of your own class! Of course, because this class is still empty, the *cannon* object that you have created doesn't have any data, properties, or behavior. We'll have to add this ourselves.

7.3.3 Adding Member Variables

The goal of the Cannon class is to group all the data and behavior that's related to the cannon game object.

In the current code, the Painter class has a few member variables associated with the cannon: one containing the cannon barrel sprite, three for the different colored discs, another for the current color, and variables for storing the position, origin, and angle of the cannon barrel.

Let's move all these member variables to the Cannon class now, because they all belong to the cannon *game object*. If we order and group these variables a bit differently, the result can look like this:

```
Texture2D cannonBarrel, colorRed, colorGreen, colorBlue;  
Vector2 barrelPosition, barrelOrigin, colorOrigin;  
Color currentColor;  
float angle;
```

Next, it's worth noting that the Cannon class doesn't know yet what the words Texture2D and Vector2 mean. You need to give the file *Cannon.cs* its own **using** instructions. Add the following three:

```
using Microsoft.Xna.Framework;  
using Microsoft.Xna.Framework.Content;  
using Microsoft.Xna.Framework.Graphics;
```

This way, Visual Studio (and the compiler) will at least recognize Texture2D and Vector2 as concepts defined by the MonoGame engine. (By the way, you can also let Visual Studio find these missing **using** instructions automatically, by hovering your mouse over the unknown word and choosing "Show potential fixes".)

There are now still several errors in the code, because the Painter class is still trying to use member variables that are now part of the Cannon class. So, we can't compile and run the game right now, but we'll fix that soon.

7.3.4 Adding Your Own Constructor Method

Now that you have added these variable declarations to the Cannon class, every Cannon instance that you create will contain these variables. However, you did not yet provide any information about which *values* these variables should have.

It's common to initialize these member variables as soon as you create a new instance of your class. That way, the Cannon object will have useful values right from the start. Do you remember that the **constructor** of a class is the method that gets called when you create a new instance? This means that the constructor method is a very good place for initializing your member variables.

There's already a default constructor for the Cannon class that doesn't do anything special. Let's overwrite this with a constructor of our own. To write your own Cannon constructor, add the following method to the Cannon class:

```
public Cannon()  
{  
}
```

You can recognize that this is a constructor method (and not a “normal” method) because of two reasons:

- A constructor method always has the same name as its class.
- Constructor methods never have a return type, not even **void**.

Currently, the Painter class is still trying to initialize the cannon’s member variables in the LoadContent method. This is no longer allowed, because the Cannon class is now responsible for initializing its own variables. Therefore, take all instructions from the LoadContent method that are related to the cannon. (These are all instructions except for the two lines that initialize spriteBatch and background.) Move these instructions into your Cannon constructor, like this:

```
public Cannon()
{
    cannonBarrel = Content.Load<Texture2D>("spr_cannon_barrel");
    colorRed = Content.Load<Texture2D>("spr_cannon_red");
    colorGreen = Content.Load<Texture2D>("spr_cannon_green");
    colorBlue = Content.Load<Texture2D>("spr_cannon_blue");

    currentColor = Color.Blue;

    barrelOrigin = new Vector2(cannonBarrel.Height, cannonBarrel.Height) / 2;
    colorOrigin = new Vector2(colorRed.Width, colorRed.Height) / 2;
    barrelPosition = new Vector2(72, 405);
}
```

In the code above, we’ve changed the order of the instructions a bit: we now first load all sprites and then initialize the positions. You could also have kept the order as it was, though.

7.3.5 Adding a Parameter to the Constructor

There’s one last problem with this constructor method: it doesn’t know what the words Content.Load mean. Load is a method of the ContentManager class defined by MonoGame. To call this method, we need an *object* that has ContentManager as its type (i.e., we need an *instance* of the ContentManager class).

In the Painter class, we had such an object: it was named Content, and we got it for free because Painter was a specific version of the Game class. However, in our new Cannon class, we don’t have this luxury, so we need to find a way to still use this Content object somehow.

The easiest solution is to add the game’s ContentManager object as a *parameter* to the constructor. That way, the constructor can use it like a variable, and it can call the Load method. Change the header of the constructor into this:

```
public Cannon(ContentManager Content)
```

This will give the constructor method an extra variable that it can use. Its name is Content and its type is ContentManager. Thanks to this extra variable, the compiler will finally understand what Content.Load means.

Now go back to the Painter class. The line `cannon = new Cannon();` is no longer allowed there, because the Cannon class no longer has a constructor with zero parameters. Writing your own constructor has caused the default constructor to disappear!

To fix this error, change the line `cannon = new Cannon();` into this:

```
cannon = new Cannon(Content);
```

What does this change mean? Well, you're now taking the `Content` object of the `Painter` class, and you're passing it to the `Cannon` constructor as a parameter. This way, the `Cannon` constructor will (indirectly) have access to the same `Content` object!

In general, whenever a method doesn't have access to a certain object, it's very common to add that object as a method parameter. You'll see many more examples of this throughout the book.

Multiple Constructors — A class can have more than one constructor method. This can be useful when you want to offer different ways of constructing an instance of your class. For example, `MonoGame`'s `Vector2` has a constructor with two parameters (which you've seen before), but also a constructor with a single parameter (which assigns that parameter to both the x - and y -coordinates), and even a version with zero parameters (which initializes x and y as 0).

You can give your own classes (such as `Cannon`) multiple constructors as well. However, you're not allowed to write two different constructors that have exactly the same number of parameters with the same types. Otherwise, the compiler won't know which constructor to use when you create an instance.

Finally, remember that you don't *have* to write your own constructor of a class. If you don't write a constructor yourself, the compiler will automatically add a default constructor with zero parameters. But as soon as you want your constructor to do something extra, you'll have to write it yourself (and then the default constructor won't exist anymore).

That's it: you've now filled in the constructor of your `Cannon` class so that it gives all member variables a proper value. Furthermore, you've given your constructor a parameter so that it can temporarily "borrow" the `ContentManager` of the `Painter` class.

By the way, we're not giving the `angle` variable a value yet. But for member variables of type `float`, the compiler automatically assigns a default value of 0, which is good enough for now. We're going to change this value in the first frame of the game anyway!

Default Values for Member Variables — When you create a new instance of a class, the constructor is the place for giving member variables a meaningful first value. For all member variables that you *don't* initialize yourself, the program will assign a *default value*. This is happening for the `angle` variable right now. In C#, the default value is 0 for numbers, **false** for Boolean variables, and **null** for class types (such as `Cannon`), but **null** is a special keyword that we'll discuss more in the next chapter.

Be aware: default values are filled in for *member* variables, but not for *local* variables inside a method. You always need to initialize a local variable yourself, before you can use it in the rest of your method.

7.3.6 Adding Other Methods

Just like in the Painter class, it is possible to add your own methods to the Cannon class. A very useful method to add is one that resets a game object to its initial state. When the player restarts the game (or a level in a game), you simply reset all the game objects to their initial states by calling the reset method on each game object. Let's add a method called `Reset` to the Cannon class.

The `Reset` method doesn't have any parameters. It simply sets the member variables in our class to some initial value. You do not have to reset all the member variable values, only the ones that will be changed while playing the game. For example, you don't have to update the `cannonBarrel` variable, since it will not change while playing the game. The `angle` variable, however, *will* change, so you have to reset it. Also, you should reset the `color`, since that may also change while playing the game. As a result, add the following `Reset` method to the Cannon class:

```
public void Reset()
{
    currentColor = Color.Blue;
    angle = 0.0f;
}
```

This method should be **public**, because we want to allow other classes to reset the cannon as well. Also, the method should have the return type **void**, because it only does a few things without giving a particular expression back.

The `Reset` method is an example of a method that *changes* the object that it works on. This is the general goal of methods: to define behavior that modifies the member variables, to give some meaningful information back to the caller of the method, or both. Observe that methods can manipulate multiple member variables. For example, the `Reset` method changes the color of the cannon, as well as the angle of the barrel.

Next to the `Reset` method, we're also going to make the cannon responsible for *drawing* itself on the screen, in the desired color. To do this, add a method called `Draw`, with the following header:

```
public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
```

Go to the Painter class and take all instructions that draw the cannon. Move these instructions to your `Draw` method of the Cannon class. The result should look something like this:

```
public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    spriteBatch.Draw(cannonBarrel, barrelPosition, null, Color.White,
        angle, barrelOrigin, 1f, SpriteEffects.None, 0);

    // determine the sprite based on the current color
    Texture2D currentSprite;
    if (currentColor == Color.Red)
        currentSprite = colorRed;
    else if (currentColor == Color.Green)
        currentSprite = colorGreen;
    else
        currentSprite = colorBlue;

    // draw that sprite
    spriteBatch.Draw(currentSprite, barrelPosition, null, Color.White,
        0f, colorOrigin, 1.0f, SpriteEffects.None, 0);
}
```


Notice how we've given the Draw method of Cannon a second parameter: `spriteBatch`, with the data type `SpriteBatch`. This is for the same reason we've also added a parameter to the constructor: the Cannon class doesn't have access to a `SpriteBatch` object, but the Painter class does. Therefore, we should pass it as a parameter, so that Cannon can temporarily borrow this object from Painter.

Remember that this Draw method isn't going to call itself automatically. You have to call it from somewhere else in the program. In other words, you've now defined how a Cannon object *can* draw itself, but you haven't yet told the program to actually *draw* the cannon instance of your game.

To draw the cannon, go back to the Painter class and change its Draw method as follows:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.White);
    spriteBatch.Begin();
    spriteBatch.Draw(background, Vector2.Zero, Color.White);
    cannon.Draw(gameTime, spriteBatch);
    spriteBatch.End();
}
```

This method first starts the sprite batch and then draws the background, as usual. It also ends by calling `spriteBatch.End()`, as usual. But in-between, it now tells the Cannon instance (stored in the `cannon` variable) to draw itself. This is a *method call* to your own Draw method that you've just created!

In summary, you've made the Cannon class responsible for loading its own sprites, for resetting its variables, and for drawing itself. Each of these methods will work on a specific instance of the Cannon class. In this case, you're using `cannon`, the member variable of Painter that contains the (only) cannon instance of the game.

You're slowly working towards a software design where each game object deals with its own tasks. In larger game projects, it's very common that every object just does its own work and that different objects hardly ever need to bother each other.

7.4 Accessing the Data in the Cannon Class

Up until now, you've seen how to define a class (Cannon) that describes a certain type of object in the game. This class has member variables that store the data belonging to your game object. The methods of the Cannon class can use this data. For example, the Reset method assigns initial values to the member variables, and the Draw method uses these member variables to draw the object.

However, sometimes you need to access an object's data directly from other classes. For example, you may want to retrieve the current position of the cannon, know what the color of the cannon is, or *change* this color based on keyboard input. These things are not yet possible right now, because nobody other than Cannon is allowed to touch the cannon's data.

Related to that, there are still some errors in the HandleInput method of Painter. This method is currently trying to use the variables `currentColor`, `barrelPosition`, and `angle`. But these variables no longer exist in the Painter class, because we've moved them to the Cannon class!

Based on what you've learned so far, you might expect that you can just replace `currentColor` by `cannon.currentColor`, replace `barrelPosition` by `cannon.barrelPosition`, and replace `angle` by `cannon.angle`. This will tell the compiler that you're referring to member variables of the cannon object.

But unfortunately, it turns out that this isn't enough to please the compiler. This is because Cannon is currently very protective about its own member variables. If you try to use such a variable in another class (via `cannon.`), Visual Studio will tell you that the variable is "inaccessible due to its protection level."

By default, member variables (and methods) are *private*: they're only accessible to methods that are inside the same class. For instance, you can access the color variable in the `Reset` method of `Cannon`, but the same variable is not accessible in the `HandleInput` method of `Painter`. In general, this protection mechanism is a good thing: it prevents objects from (possibly) messing up other objects everywhere in the code. This forces programmers to think about how objects share their information.

How can we change this? Well, there are (at least) three different ways to make the data of `Cannon` more accessible. This section will give you these solutions, starting with the easiest one and ending with the nicest one in terms of programming style. Very soon, you'll finally be able to run the `Painter` game again.

7.4.1 Option 1: Marking Member Variables as *public*

The first (and easiest) option is to change the accessibility of `Cannon`'s member variables. Just like with methods, you can use **access modifiers** to define where you're allowed to use these variables. As we said before, the member variables of a class are *private* by default (just like methods). If you write the word **public** in front of a variable declaration, then that variable becomes accessible to other classes. For instance, if you replace this declaration:

```
Color currentColor;
```

by this:

```
public Color currentColor;
```

other classes will be allowed to refer to the member variable `currentColor` of a `Cannon` object. This way, the expression `cannon.currentColor` in the `Painter` class will be allowed again.

So, the easiest way to get rid of the compiler errors is to write **public** in front of all your methods and member variables. This allows you to do everything with every class at every point in your code.

But from a design point of view, this isn't a very nice thing to do. Think of it this way: you've created different classes so that each class is responsible for its own data, but now you're allowing classes to change *each other's* data as well! In a way, this beats the purpose of splitting your code into self-contained parts.

A better option is to think more carefully about *what* you allow other classes to do with your class. This brings us to the second and third options.

7.4.2 Option 2: Adding Getter and Setter Methods

The second option is to make the (private) member variables available via (public) *methods*. Programmers often refer to these kinds of methods as *getters* (for retrieving variables) and *setters* (for updating variables). It's common practice to let the names of such methods start with `Get` and `Set`, to make extra clear what these methods do.

For instance, you could add a `GetPosition` method to the `Cannon` class, which provides the caller with the position of the cannon:

```
public Vector2 GetPosition()
{
    return barrelPosition;
}
```

This way, the Cannon class can still protect the `barrelPosition` member variable. If you now replace `cannon.barrelPosition` by `cannon.GetPosition()`, you'll call the getter method instead of trying to use the member variable directly.

You could also add a `SetPosition` method that (indirectly) changes the `barrelPosition` variable, but you don't need it in this case: the game never needs to change the barrel's position. This is another advantage of this solution: you can decide for yourself which variables you may *get* and which variables you may *set*.

The `currentColor` member variable is an example of a variable that we want to *set*: we want to change this color based on keyboard input. So, you can give the Cannon class a `SetColor` method, which takes a color as a parameter and updates the `currentColor` variable accordingly:

```
public void SetColor(Color col)
{
    currentColor = col;
}
```

A nice bonus of a setter method is that you can add *extra code* that checks the parameter value. For instance, a Cannon object should only be allowed to have the colors red, green, and blue. If somebody ever tries to set a different color than those three, we can use an **if** instruction to prevent the color from changing:

```
public void SetColor(Color col)
{
    if (col != Color.Red && col != Color.Green && col != Color.Blue)
        return;
    currentColor = col;
}
```

This version of the `SetColor` method uses a **return** instruction to “jump out” of the method (as explained earlier in this chapter) if the given color is incorrect.

You might wonder why this is useful. After all, you are the one writing the code, so you should know which colors (not) to use, right? That's true, but if you plan to work in a team with other developers later on (which is quite normal in the game industry), then the programmers that use your classes may accidentally do things that they're not supposed to do. By building these kinds of securities into your classes, you know for certain that (in this case) the cannon will never be any other color than red, green, or blue.

One “problem” of this approach is that you can't warn other developers about which colors to use. You could improve this by adding comments to the code and hoping that the other developers will read these comments. Another option is to build in *exception handling*, but we won't go into that topic until the very last part of this book.

In the case of Cannon, you'd currently need three of these getters and setters: `GetPosition`, `SetColor`, and `SetAngle`. It's good practice to only give a class the getters and setters that it really needs. So start with only private member variables, and discover later which of these variables you want to get or set from outside your class. And whenever you think you need a setter, think carefully if you *really* need it.

7.4.3 Option 3: Adding Properties

In many object-oriented programming languages, getter and setter methods are the only way to access nonpublic data inside an object. This means that programmers have to write separate getters and setters

for each member variable that requires it. Because this is a pretty tedious task that programmers need to do so often, there's another option that exists specifically in the C# language: **properties**.

Roughly speaking, a property is a shorter replacement for a getter and a setter method. It defines what happens when you retrieve data from an object and what happens when you assign a value to data inside an object.

To show how this works, let's first convert the `GetColor` method into a property, called `Color`:

```
public Color Color
{
    get { return currentColor; }
}
```

The syntax of properties is different from what you've seen before. Just like with methods, you start with an access modifier (**public**) and a data type (`Color`), followed by a name you can choose yourself. In this case, we've chosen the name `Color`—this happens to be the same name as the `Color` data type, but that's allowed in C#. The compiler is smart enough to know if you mean the `Color` data type or the `Color` property. (But if this confuses you, feel free to give the property another name!)

The property name should be followed by two curly braces. Between these braces, you can write the keyword **get** followed by another pair of curly braces. This part is basically the replacement of a getter method. Inside the **get** block, you should use the keyword **return** followed by the expression you want to return. In the case of the `Color` property, we immediately return the (private) member variable `currentColor`. But just like a getter method, a **get** block could also be more complicated, for example, with **if** instructions and multiple **return** keywords. You won't have to do that very often, though.

You can now write the expression `cannon.Color` in the `Painter` class. This is an expression of type `Color`, and the program will get its value by going into the **get** block of your `Color` property. This is very similar to how the program would jump into a `GetColor` method.

Next to a **get** block, a property can also have a **set** block, which (you guessed it!) plays the role of a setter method.⁴ Here's the same `Color` property extended with a setter:

```
public Color Color
{
    get { return currentColor; }
    set
    {
        if (value != Color.Red && value != Color.Green && value != Color.Blue)
            return;
        currentColor = value;
    }
}
```

The **set** block is very similar to our previous `SetColor` method, but with one important difference: we've replaced the parameter name `col` by the keyword **value**. This is a special C# keyword that exists only in a **set** block of a property; it indicates the *parameter* of the setter. In other words, the `SetColor` method had a single parameter that we could name ourselves, but in a property, this parameter always has the name **value**.

You can now write an instruction like `cannon.Color = Color.Red`; inside the `Painter` class. Because the expression `cannon.Color` is now on the left side of an equals sign, the compiler will know that you're trying to *assign* a value to the `Color` property. This means that it will go into the **set** block. (And if you'd

⁴You *did* guess that, right?

write the instruction `cannon.Color = Color.Black`; instead, the cannon's color will not change, because the color black will be blocked by your `if` instruction.)

You can also create properties that only have a **get** block, if you never want to change them from outside the class. Simply leave out the **set** block and you're done. A nice example for this is the position of the barrel. You could create the following property for it:

```
public Vector2 Position
{
    get { return barrelPosition; }
}
```

If a property only contains a **get** block, it's called a **read-only property**, meaning that you can only *read* the data and not modify it. For example, the following instruction would result in a compiler error (feel free to try it):

```
cannon.Position = Vector2.Zero;
```

Finally, let's create a third property for the cannon's angle:

```
public float Angle
{
    get { return angle; }
    set { angle = value; }
}
```

This makes the angle member variable accessible via an Angle property. Admittedly, it's not that different from making the angle variable public: every class can now change this variable. But if we ever want to make the **set** block more complicated (e.g., to only allow angles within a certain range), then the property structure is already there for us.

In summary, properties are nice replacements of getter and setter methods. You can use them almost like a member variable, and the compiler will then automatically know if it should use the **get** block or the **set** block.

When you want to make private member variables accessible to other classes, it's really a matter of taste whether you use *properties* or *methods*. Many C# programmers prefer properties, because properties are a very specific feature of the C# language. It's one of the key features that is "missing" from Java, for example. But no matter what your preference is, pretty much everyone⁵ agrees that anything's better than simply making all member variables **public**.



Quick Reference: Making Data Accessible to Other Objects

By default, the member variables (and methods) of a class are private: only the class itself can use them. To make a member variable available in other classes, you can do three things:

1. Mark the variable as **public**. This is usually not preferred, because it allows you to change the variable everywhere.

(continued)

⁵... Well, except for that annoying guy at the office, but he doesn't know what he's doing.

2. Create public *methods* for retrieving (“getting”) and updating (“setting”) the variable. This gives you full control over what is allowed (and what isn’t), and it allows you to add extra checks to the setter method.
3. Create a public *property* for the variable, with separate **get** and **set** blocks. This is basically a shorter way to write getter and setter methods in C#.

Perhaps without knowing it, you’ve used properties before in earlier chapters of this book. Some examples are the `TotalGameTime` property of the `GameTime` class and the `LeftButton` property of the `MouseState` class. Back then, we told you that a property is almost the same as a member variable. Now that you’ve read this section, you’ll hopefully agree that a property is really more like a *method*, or even two methods grouped together.

7.4.4 Using the Properties of Cannon in the Painter Class

Go ahead and add the properties `Position`, `Angle`, and `Color` to the `Cannon` class. Inside the `Painter` class, you can now write `cannon.Position` to retrieve the cannon’s position, and `cannon.Angle` to retrieve or update the cannon’s angle, and `cannon.Color` to retrieve or update the cannon’s color (with extra checks that prevent you from using the wrong colors).

If you change the `Painter` class so that it uses these properties instead of the member variables, you should be able to compile and run the program again.



CHECKPOINT: Painter3

If you’ve made any mistakes along the way, or if you want to double-check your result, take a look at the `Painter3` example program of this chapter.

The game will still do the same as before, but you’ve improved the organization of the code. Instead of storing a huge list of member variables in one class, you’re now letting the cannon store its own data. You’ve also thought carefully about how you’re allowed to retrieve or change this data in other places.

7.4.5 More on Properties

The properties that you added to the `Cannon` class correspond directly to member variables: they get and set the values of the position, color, and angle variables.

However, properties don’t always *have* to correspond to member variables. A property may also use one or more member variables to calculate a *new* value. For example, you could add a property called `Bottom`, which calculates the bottom *y*-coordinate of the cannon barrel. This (read-only!) property would return a **float** value, which gets calculated every time the property is called. The header and body of this property would be given as follows:

```
public float Bottom
{
    get { return position.Y + barrelOrigin.Y; }
}
```

As you can see, this property uses the position and barrelOrigin member variables to calculate a **float** value. Of course, another solution would be to add an extra member variable to the Cannon class (called bottom) and to calculate it only once, in the constructor. The advantage of using a property is that you'll store less redundant data in your class. You only store the position of the object and the height of the cannon barrel sprite once, in the cannonBarrel and position variables. The disadvantage is that recomputing the bottom every time the property is accessed will make your code less efficient. This is a *design choice* that you have to make as a developer: it depends on what *you* think is more important.

If a property is directly linked to a member variable without any other behavior (such as in our Angle example), then you actually can write it in a shorter way that leaves out the member variable completely. Behind the keywords **get** and **set**, you can replace the {...} parts by semicolons, like this:

```
public float Angle { get; set; }
```

In this version, you can get and set the Angle property directly, and you don't need the angle member variable anymore. (In the background, the compiler will actually create a "hidden" member variable, but you don't have to worry about that.) You'll see examples of this in later chapters. Just remember that this shortcut is only possible if you require no special behavior in your **get** or **set** parts. For example, it wouldn't work for our Color property, because that property requires extra checks in its **set** component.

There are many ways to design classes and their member variables, methods, and properties. There is not one "perfect" design that caters for all possible needs. Whatever choices you make, try to make them carefully (by keeping your goals in mind), and make sure that other programmers can also understand them.

7.5 Reorganizing the Class Structure

You've now seen how to create your own class with its own member variables, methods, and properties. In this section, you'll use this knowledge to organize the Painter project even further. You'll create a separate class for input handling and a separate class that represents the game world. These elements will return in the other games of this book as well. You could almost say that we're creating our own "game engine" inside MonoGame!

We'll go through the programming process a bit faster, challenging you to fill in the details yourself. See if you can create the code yourself before you look at the answer. The result of this section can be found in the Painter4 example project.

7.5.1 A Class for Handling Input

In the Painter3 project, there are two classes: the Painter class that initializes the graphics device, loads sprites, and so on and the Cannon class that represents the cannon game object. Game objects (such as the cannon) are good candidates for separate classes, but you can use classes for other concepts as well.

First, we'll add another class that will make input handling a bit easier. Up until now, if you wanted to check if a player pressed a key, you needed to store the previous and current keyboard state and use a fairly complicated **if** instruction. For example:

```
if (currentKeyboardState.IsKeyDown(Keys.R) && previousKeyboardState.IsKeyUp(Keys.R))
{
    cannon.Color = Color.Red;
}
```

The same kind of thing is happening for mouse button presses. To make this easier, let's create a *helper class* (called *InputHelper*) that will provide a few simple but very useful methods and properties. The *InputHelper* class will have two tasks:

1. Maintain the previous and current mouse and keyboard states;
2. Provide methods and properties that allow for easy access of mouse and keyboard information.

The *Painter* class will create an *instance* of *InputHelper* and let that instance do the work.

Create a new *InputHelper* class, in a new file named *InputHelper.cs*. The class should use the libraries *Microsoft.Xna.Framework* and *Microsoft.Xna.Framework.Input*, so add two **using** instructions for that. Next, give your class the *MouseState* and *KeyboardState* member variables that *Painter* currently has. In other words, move these member variables from *Painter* to *InputHelper*.

These states need to be updated at the beginning of every update. Therefore, add an *Update* method to this class that copies the last current mouse and keyboard state to the previous state and that retrieves the current state:

```
public void Update()
{
    previousMouseState = currentMouseState;
    previousKeyboardState = currentKeyboardState;
    currentMouseState = Mouse.GetState();
    currentKeyboardState = Keyboard.GetState();
}
```

In the *Painter* class, create a member variable *inputHelper* of type *InputHelper*. In the constructor method of *Painter*, initialize it by calling the default constructor of the *InputHelper* class. You don't have to write your own constructor of *InputHelper*: we really don't need to do anything special when constructing an *InputHelper* object.

In the *HandleInput* method of the *Painter* class, you're currently starting by updating the mouse and keyboard states yourself. Replace these instructions by a single method call of the *inputHelper* object:

```
inputHelper.Update();
```

This will do the same work, but this time it will happen in a nicely detached class.

Now, let's give the *InputHelper* class a few convenient methods and properties. These will give us the information needed in the game, while hiding some of the complicated details.

First, add a method that checks if a particular keyboard key has been pressed in the current frame. This method returns a Boolean value (**true** or **false**) indicating whether or not the key has been pressed. The method has one parameter, which is the key that you want to check. That way, you can call the method with any possible key as a parameter, and the method will check that particular key. Here's what the method looks like:

```
public bool KeyPressed(Keys k)
{
    return currentKeyboardState.IsKeyDown(k) && previousKeyboardState.IsKeyUp(k);
}
```


As you can see, this method only checks if the given key is currently down *and* if it wasn't down yet in the previous frame. Now that you have this method, the **if**-instructions (back in the `Painter` class) for checking R, G, and B key presses become much simpler:

```
if (inputHelper.KeyPressed(Keys.R))
{
    cannon.Color = Color.Red;
}
else if (inputHelper.KeyPressed(Keys.G))
{
    cannon.Color = Color.Green;
}
else if (inputHelper.KeyPressed(Keys.B))
{
    cannon.Color = Color.Blue;
}
```

You can do something similar for checking if the player has clicked the left mouse button in the current frame. Let's give this method the name `MouseLeftButtonPressed`. This method also returns a value of type **bool**, but it doesn't have any parameters, because there's nothing to vary: it always concerns the left mouse button. The complete method is given as follows:

```
public bool MouseLeftButtonPressed()
{
    return currentState.LeftButton == ButtonState.Pressed
    && previousMouseState.LeftButton == ButtonState.Released;
}
```

Finally, add a property `MousePosition` that returns a `Vector2` containing the current mouse position. Since this is a read-only property, you only need to write a **get** part:

```
public Vector2 MousePosition
{
    get { return new Vector2(currentMouseState.X, currentMouseState.Y); }
}
```

In the `Painter` class, you can now replace the expression `currentMouseState.X` by the expression `inputHelper.MousePosition.X` (and make a similar change for `Y`). The compiler should now be satisfied: you can compile and run the program again.



CHECKPOINT: Painter3b

You now have a simple `InputHelper` class that updates the mouse and keyboard states and that provides a few convenient methods and properties for accessing the information.

In more complicated games, you can imagine that this class needs to become much bigger. You could add many useful things, such as handling dragging, dropping, selecting, and detecting special key combinations such as “Ctrl-S”. The class could even take care of managing other input devices such as a multi-touch screen or a gamepad.

However, for the `Painter` game, the current version of `InputHelper` is sufficient. In the rest of this book, you'll keep using and improving this class.

7.5.2 Letting the *Cannon Class Maintain Itself*

The Cannon class is already responsible for drawing its own sprites. Let's take this a step further so that Cannon also takes care of its own input handling. To do this, give the class a new method with the following header:

```
public void HandleInput(InputHelper inputHelper)
```

Move all lines from Painter's HandleInput method to this new method, except for the first instruction (inputHelper.Update();). To satisfy the compiler, the Cannon class will now also have to include the System and Microsoft.Xna.Framework.Input libraries. You should now also replace cannon.Color by Color, cannon.Position by Position, and cannon.Angle by Angle, because you're now inside the cannon object itself. That is, you don't need a reference to a specific cannon anymore, because the cannon itself is already that instance.

The HandleInput method of Painter now only has two jobs: to update the inputHandler object and to make sure that the cannon handles its own input. Therefore, this method should contain only two instructions:

```
inputHelper.Update();
cannon.HandleInput(inputHelper);
```

Note that this is another example of passing a useful object as a parameter. The Painter class stores an InputHelper object, but the Cannon class does not. To let a method of Cannon use this InputHelper object nonetheless, you pass it as a parameter.

In fact, the HandleInput and Update methods of the Painter class have now become so simple that we can merge them into one method. You can remove the HandleInput method and just write everything in Update, as follows:

```
protected override void Update(GameTime gameTime)
{
    inputHelper.Update();
    cannon.HandleInput(inputHelper);
}
```

If you look critically at the Cannon class now, you'll notice that the properties Position, Angle, and Color are now only used inside the Cannon class itself. By making the cannon completely responsible for its own data, it has become unnecessary to expose this data via (public) properties. This means that you can now mark these properties as **private**! The properties themselves are still nice to have, for example, because of the color checks in the Color setter.

At this point, it's really up to you how protective you want the Cannon class to be. In the Painter3c example of this chapter, we've made the Angle property private, because we expect that other classes are no longer interested in this angle. We've kept the Position property public: even though it's not used anywhere right now, we can imagine that other classes may be interested in the cannon's position. The Color property is a special case: we've made the getter public and the setter private! This is something haven't talked about yet: you can give the **get** and **set** parts of a property different **access modifiers**. You can do this by writing another access modifier before the keyword **get** or **set**. Here's what the Color property looks like:

```
public Color Color
{
    get { return currentColor; }
    private set
    {
```

```

    // code that updates the color
}
}

```

So, the property as a whole is still publicly accessible, but the **set** part can only be used inside the Cannon class itself.



CHECKPOINT: Painter3c

Compile and run the game again. You've now created an even "cleaner" version of the program!

As you've noticed, we're constantly moving things around in our example code. This may seem chaotic, but we're actually doing it on purpose, to show you that *it's okay* to change the layout of your program later on. In the programming courses we've taught, we've noticed that students are often scared to change the responsibilities of their classes and objects. This is something that you really don't have to be afraid of, as long as you know what you're doing. By showing you our examples, where we deliberately improve the program step by step, we hope that you'll get a good feeling for it.

7.5.3 A Class for Representing the Game World

The final class that you're going to create in this chapter is called `GameWorld`. It will be responsible for managing all game objects: creating them, updating them, and drawing them. Currently, the game world only consists of a background sprite and a cannon, but (again) you can imagine that this will become more complicated later on. So, even though creating a separate `GameWorld` class might seem like overkill at the moment, you'll benefit from it in later chapters.

Create a `GameWorld` class and let it include three libraries:

```

using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

```

Next, give it the two member variables that are currently related to the game world (and remove these member variables from the `Painter` class):

```

Texture2D background;
Cannon cannon;

```

The `GameWorld` class will be in charge of creating, updating, and drawing these objects. The first step (creating the objects) will be done in the constructor method. Create your own `GameWorld` constructor and give it the following two instructions:

```

background = Content.Load<Texture2D>("spr_background");
cannon = new Cannon(Content);

```

Note that the constructor of `Cannon` requires a variable of type `ContentManager`, which you don't have yet in the `GameWorld` class. To give your method access to such a variable, you can use the same trick as before: give the `GameWorld` constructor a *parameter* of type `ContentManager`, so that all the constructors in your game can pass that object around. The header of the constructor then becomes:

```

public GameWorld(ContentManager Content)

```

In the Painter class, you should now define a member variable `gameWorld` that will contain the (only) instance of the `GameWorld` class. This leaves you with only a few member variables in the Painter class:

```
GraphicsDeviceManager graphics;
SpriteBatch spriteBatch;
InputHelper inputHelper;
GameWorld gameWorld;
```

As a recap, the `graphics` variable contains the graphics device manager as usual, the `spriteBatch` variable is used for drawing sprites, and the `inputHelper` object has a few convenient methods and properties for input handling. In the `LoadContent` method, initialize the `gameWorld` variable as follows:

```
gameWorld = new GameWorld(Content);
```

This will call the `GameWorld` constructor, which will (in turn) load the background and create the cannon object. You can remove the other instructions from `LoadContent` now, because our `GameWorld` constructor has taken them over.

There are basically three things that the `gameWorld` object should do to manage the game objects: make sure that the game objects handle player input, make sure that they're updated, and finally draw them on the screen. You'll now give the `GameWorld` class three extra methods for this: `HandleInput`, `Update`, and `Draw`.

The `HandleInput` method should tell all game objects to handle their own input. Currently, the only game object that can do this is the cannon, so this method will contain only one instruction. Also, the method should take an `InputHelper` object as a parameter again, because it needs to pass that same object to the cannon. Here's the full method:

```
public void HandleInput(InputHelper inputHelper)
{
    cannon.HandleInput(inputHelper);
}
```

The `Update` method should tell all game objects to update themselves. Currently, none of our game objects actually do any updating, so this method can be empty. However, to prepare the game for future versions, we'll already give this method a `GameTime` parameter:

```
public void Update(GameTime gameTime)
{
}
```

The `Draw` method should first draw the background and then tell the other game objects to draw themselves. The `Cannon` class already has its own `Draw` method that we can call here. Because that method already required a `GameTime` parameter *and* a `SpriteBatch` parameter, our new `Draw` method of `GameWorld` needs these parameters as well.

We've chosen to let this `Draw` method do all the work *except* clearing the screen: that's the only task that will still be done by the main Painter class. In short, the `Draw` method of `GameWorld` looks like this:

```
public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    spriteBatch.Begin();
    spriteBatch.Draw(background, Vector2.Zero, Color.White);
    cannon.Draw(gameTime, spriteBatch);
    spriteBatch.End();
}
```

Once you've made all these changes, the Painter class itself can be simplified a lot. It should only initialize some objects when the game starts, and it should clear the screen in the Draw method. Apart from that, it passes all the work to the gameWorld member variable, by calling gameWorld.HandleInput, gameWorld.Update, and gameWorld.Draw. The full Painter class is displayed in Listing 7.1. Have a look at that class and see if you understand the final changes that you need to make.

Listing 7.1 The Painter class after you've created a separate class for the game world

```

1  using Microsoft.Xna.Framework;
2  using Microsoft.Xna.Framework.Graphics;
3  using System;
4
5  class Painter : Game
6  {
7      GraphicsDeviceManager graphics;
8      SpriteBatch spriteBatch;
9      InputHelper inputHelper;
10     GameWorld gameWorld;
11
12     [STAThread]
13     static void Main()
14     {
15         Painter game = new Painter();
16         game.Run();
17     }
18
19     public Painter()
20     {
21         Content.RootDirectory = "Content";
22         graphics = new GraphicsDeviceManager(this);
23         IsMouseVisible = true;
24         inputHelper = new InputHelper();
25     }
26
27     protected override void LoadContent()
28     {
29         spriteBatch = new SpriteBatch(GraphicsDevice);
30         gameWorld = new GameWorld(Content);
31     }
32
33     protected override void Update(GameTime gameTime)
34     {
35         inputHelper.Update();
36         gameWorld.HandleInput(inputHelper);
37         gameWorld.Update(gameTime);
38     }
39
40     protected override void Draw(GameTime gameTime)
41     {
42         GraphicsDevice.Clear(Color.White);
43         gameWorld.Draw(gameTime, spriteBatch);
44     }
45 }
```



CHECKPOINT: Painter4

Congratulations: you’ve organized your first project into multiple classes! As you can see, the individual methods of Painter, GameWorld, and Cannon are now quite short. The Cannon class contains the most “intelligence,” because it actually does something with the player’s input. The Painter and GameWorld classes mostly tell other objects to do the dirty work for them.⁶ This is very common in object-oriented programming: all classes are nicely isolated pieces of code, each with their own data and behavior.

7.6 What You Have Learned

In this chapter, you have learned:

- how to organize instructions into methods, possibly with parameters and/or return values;
- how to organize methods and member variables into classes;
- that a class is a blueprint for objects;
- how to use the keywords **public** and **private** to change the accessibility of member variables and methods;
- how to use properties (or getter and setter methods) to let other classes use private member variables indirectly;
- that **this** refers to the object that you are currently manipulating in a method or property;
- how to organize a game project into multiple classes, where each class represents a certain (type of) game object or a certain kind of work.

7.7 Exercises

1. *Keywords*

This question tests your knowledge of several C# keywords.

- a. What does the word **void** mean, and when do we need this keyword?
- b. What does the word **int** mean, and when do we need this keyword?
- c. What does the word **return** mean in a C# instruction, and when do we need it?
- d. What does the word **this** mean in a C# instruction, and when do we need it?

2. *Properties in Painter*

This question lets you change the properties in the Painter program (and add new properties to it).

- a. Look at the Angle property that you’ve added to the Cannon class. How could you change the *setter* of this property to only allow angles between 0 and 2π radians?
- b. Can you add a property to the Cannon class that allows reading the cannon barrel sprite?
- c. Can you add a property to the Painter class that gives access to the `spriteBatch` object?

3. *Methods with a Result*

In this question, you’ll write methods that take **float** parameters and return another **float** value.

- a. Write a method `Circumference` that gives as a result the circumference of a rectangle, whose width and height are given as parameters.

⁶Sounds like a cool job! Personally, we’d rather be a game world than a cannon, at this point.

- b. Write a method `Diagonal` that gives as a result the length of the diagonal of a rectangle, whose width and height are given as parameters. *Hint:* Use the `Math.Sqrt` method for computing the square root of a number.

4. * *Methods About Dividers*

In this question, you'll write several methods that take **int** parameters and return a **bool** value.

- a. Write a method `Even` which indicates whether a number passed as a parameter is an even number. Determine what the best type is for the parameter and the return value.
- b. Write a method `MultipleOfThree` that indicates whether its parameter is a multiple of three.
- c. Write a method `MultipleOf` with two parameters x and y that determines if x is a multiple of y .
- d. Write a method `Divisible` with two parameters x and y that determines if x is divisible by y (so x/y should have no remainder).

5. * *Multiple Flying Balloons*

In the exercises from Chap. 6, you've updated the `FlyingSprites` example such that balloon bounces off the edge of the screen. Let's improve this program by using classes and objects.

- a. Move the behavior of the balloon to a separate `Balloon` class. Give the overall game class an instance of your `Balloon` class, and check that the game still behaves as expected.
- b. Modify the constructor of `Balloon` to include a starting position and a starting velocity.
- c. Add multiple instances of `Balloon` to the game, with different starting positions and velocities. Do you notice how the `Balloon` class avoids a lot of copying and pasting of code?

Chapter 8

Communication and Interaction Between Objects



The previous chapter has laid out the basic class structure for the Painter game. You've created a separate Cannon class that represents the cannon object in the game. You've also defined a simple input helper class and a class for representing the game world. You have already filled part of that game world by adding a background image and a cannon. The cannon handles its own input to change its color and angle.

In this chapter, we'll first look at how objects are stored in memory. You'll learn about the difference between primitives, classes, and structs in this context. Next, you will add the other game objects (the ball and the paint cans) to the game, and you will program how they interact with each other. For both types of game objects, you will add a new class to the program. Along the way, you'll learn how to use the keyword **static** to make certain data available everywhere in your program, how to use the Random class to add randomness to your games, and how to use the Rectangle struct to check if two objects are overlapping.

This will be another long chapter: we want to give you a good picture of what it's like to create a game in which multiple classes communicate. In the rest of this book (and in the rest of your programming career), you'll basically keep repeating this same concept, but for increasingly complicated programs.

At the end of this chapter, the Painter game will be largely finished: you can shoot balls that are subject to gravity, and these balls can change the color of a paint can that they touch.

8.1 How Objects Are Stored in Memory

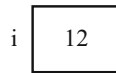
First, let's have a look at how objects and variables are actually dealt with in the computer's memory. You've seen in Chap. 4 that there are **primitive data types** such as **int** or **float**. There are also **class types** used for the class instances in your program. When it comes to memory, primitive types and class types are handled differently.

8.1.1 Primitive Types: Values

Variables of a primitive type are directly associated with a place in memory. For example, consider the following instruction:

```
int i = 12;
```


After this instruction has been executed, the memory will (schematically) look like this:



That is, *i* refers immediately to a place in memory, and the number 12 is stored at that place. Now, you can create a second variable *j*, and store the value of variable *i* inside that variable:

```
int j = i;
```

The memory will then look like this:



Both variables store their *own* “copy” of the number 12. Therefore, if you assign another value to the variable *j*, for example, by executing the instruction *j* = 24;, this is what happens in memory:



The variable *j* will have a new value (as expected), but the variable *i* will still have the value 12. This effect holds for all primitive types: if you let multiple variables take each other’s values, then each variable will actually store a *copy* of that value. Changing a variable’s value will only change that one copy.

8.1.2 Class Types: References

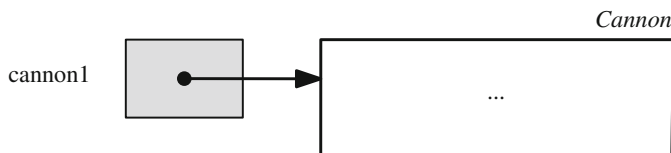
Class types work differently. Recall that every instance of a class *MyClass* will have the data type *MyClass*. Because a class instance can store many member variables, copying the entire object could cost quite some time and memory. For that reason, C# chooses *not* to copy these objects by default.

If you create an instance *myInstance* of class *MyClass*, then the variable *myInstance* will not store this entire instance directly. Instead, the instance itself will be stored at a different place in memory, and *myInstance* will only store a *reference* to that place.

As an example, consider the following instruction that creates a *Cannon* object:

```
Cannon cannon1 = new Cannon(Content);
```

After this instruction, the memory looks like this:

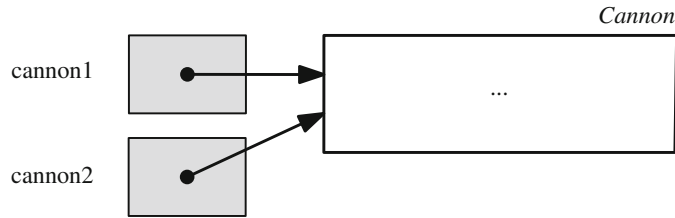


In this image, we use an arrow to visualize that *cannon1* stores a reference to an object and not the object itself. This is the main difference compared to variables of *primitive* types.

Let’s say you now declare another variable, *cannon2*, and you assign the value of *cannon1* to it:

```
Cannon cannon2 = cannon1;
```

After this instruction, the memory looks like this:



In other words, the `cannon2` variable will store a second reference to the same place in memory that `cannon1` is already referring to. There is still only a single `Cannon` instance, but there are now two variables pointing to it. We haven't created any copies.

As a result, if you would change the color of the cannon as follows:

```
cannon2.Color = Color.Red;
```

then afterwards, the expression `cannon1.Color` will *also* be `Color.Red`, since both `cannon1` and `cannon2` refer to the same object! This may sound annoying, but it usually works in your advantage: you can let multiple variables *share* the same object, instead of having two copies that you need to keep in sync all the time.

If you want `cannon1` and `cannon2` to refer to two *different* `Cannon` objects, you'll have to create these two objects yourself. For example:

```
Cannon cannon1 = new Cannon(Content);
Cannon cannon2 = new Cannon(Content);
```

That way, there will be two distinct `Cannon` instances in memory, and both variables will refer to a different instance.

8.1.3 The Consequence for Method Parameters

This difference between *values* and *references* also has an effect on how objects are passed around in methods. For example, in the `Draw` method of the `Cannon` class, you pass a `SpriteBatch` object as a parameter. Because `SpriteBatch` is a class type, you pass this parameter as a *reference*. In total, there's only a single `SpriteBatch` object that is being passed around by all `Draw` methods.

The result of this is that you also can *change* the `SpriteBatch` object in your method. In fact, the method call `spriteBatch.Draw` does exactly that!

We say that class objects are *passed by reference*. Hypothetically, you could imagine a method that takes a `Cannon` parameter and changes the color of that cannon:

```
void ChangeColor(Cannon cannon)
{
    cannon.Color = Color.Red;
}
```

(assuming that the `Color` property has a public setter again.) If you then execute the following instructions:

```
Cannon cannon1 = new Cannon(cannonSprite);
ChangeColor(cannon1);
```

the object referred to by `cannon1` will receive a red color. Even though the method parameter `Cannon cannon` is a different variable than `Cannon cannon1`, they both refer to the same location in memory. Therefore, changing the `cannon` object to which `cannon` refers will also change the `cannon` to which `cannon1` refers.

By contrast, if you pass parameters of a primitive type such as **float**, then these will be *passed by value*. You can probably guess what that means. A method parameter of a primitive type will actually store a *copy* of the value that you give to it. To see this in action, consider the following (incorrect) method that tries to calculate the square of its parameter:

```
void Square(float f)
{
    f = f * f;
}
```

Next, consider the following instructions:

```
float someNumber = 10.0f;
Square(someNumber);
```

You're passing a value of 10.0f to the Square method here. However, inside the Square method, the parameter **float** *f* will store a new copy of the number 10.0f. Unlike with class types, it's *not* a reference to the original number that *someNumber* is storing. In other words, *f* and *someNumber* correspond to different locations in memory.

Therefore, the Square method calculates the square of a new number and doesn't do anything with the result. After the method call, the variable *f* will go out of scope, and the variable *someNumber* will still store the value 10.0f (and not 100.0f).

References to Primitives — In case you're wondering, there's also a way to pass primitive types by *reference*. There's a keyword **ref** that you can add in front of a method parameter:

```
void Square(ref float f)
{
    f = f * f;
}
```

And when you call such a method, you have to (again) state explicitly that you want to pass your variable as a reference.

```
float someNumber = 10.0f;
Square(ref someNumber);
```

That way, the variable *someNumber* will actually store the value 100.0f after the method call, as desired.

We don't recommend doing this too often, though. Usually, when you're using the keyword **ref** in this way, it's a sign that your method should simply have a return value. For instance, the Square method should just return a new **float** value instead of trying to change the original value:

```
float Square(float f)
{
    return f * f;
}
```

This version of Square is also much easier to understand. For now, we suggest that you avoid using the **ref** keyword yourself. But if you ever find it in someone else's code, you will at least know what it means.

8.1.4 The *null* Keyword

So, if variables of class types contain *references* to objects, instead of direct values, what does such a variable refer to *before* you assign it any value? Suppose that you declare a member variable of type `Cannon`:

```
Cannon anotherCannon;
```

At this point, you have not yet created an object (using the **new** keyword) that this variable points to. The memory currently looks like this:

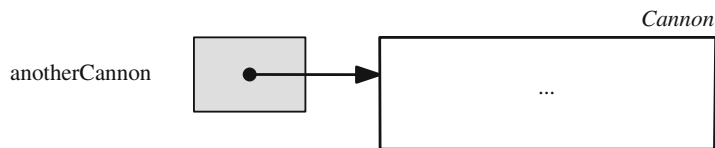


The variable is not yet pointing to anything. In C# (and in many other programming languages), this “nothingness” is indicated by the keyword **null**.

It is even possible to check in your C# program whether a variable is pointing to an object or not. You can do this by checking if the reference in a variable is equal to **null**. Here’s an example:

```
if (anotherCannon == null)
    anotherCannon = new Cannon(cannonSprite);
```

This code checks if the variable is equal to **null** (in other words: not pointing to any object yet). If so, we create a `Cannon` instance using the **new** keyword. After that, the memory will look like this:



You can also *assign* a value of **null** to a variable:

```
anotherCannon = null;
```

That way, you will “destroy” the reference that `anotherCannon` used to have. In terms of our example images, you “remove the arrow” from `anotherCannon` to its associated object. But watch out: this does not automatically *destroy* the object that it referred to, because there might still be other variables (other “arrows”) referring to it.



Quick Reference: Variables, References, and null

A variable of a *primitive type* corresponds directly to a place in memory. If you copy it to another variable, or pass it as a method parameter, then you will create a new version of that value. This is called *passing by value*.

A variable of a *class type* doesn’t directly store an object. Instead, it stores a reference to the place in memory where the actual object is stored. If you copy it to another variable, or pass it as a method parameter, then you will create a new reference to the same object. This is called *passing by reference*.

If a variable of a class type does not refer to anything, it will store the special value **null**. For example, this is the case *before* you’ve initialized such a variable. You can also set the variable back to **null** yourself.

8.1.5 Reference Counting and Garbage Collection

So, variables of class types store references to actual objects in memory, and you can have multiple references to the same object. You can also remove these references again, by letting a variable refer to something else or by setting a variable to **null**.

If there are no more references to an object, then your C# program will automatically recognize that this object can be removed from memory. This is called **garbage collection**: every once in a while, the program will remove any unreferenced objects from memory, so that this memory can be used for other things again. The program will decide for itself *when* it cleans up the memory. It won't always happen immediately, but it will happen at some point.

Garbage collection is something that C# automatically does for you. This is one of the key things that makes C# a nice first programming language to learn. The language Java also has garbage collection, but (for example) C++ does not. In a C++ program, you are in charge of managing the memory yourself, which can lead to all sorts of problems if you don't know what you're doing.

Memory management will probably be the biggest hurdle if you ever switch from C# to C++. On the other hand, once you know how it works, it's not *that* scary anymore, and it actually allows you to create very efficient programs! This is why many AAA games are still being developed in C++.

In this book, we won't go into the details of memory management any further. Just enjoy the fact that you don't have to do it yourself!

On the other hand, you should be aware that objects stay in memory as long as there are still references to it. This won't be a problem in the small games of this book. But in very large projects, if your game starts to consume too much memory, it might be a sign that you still have to set some variables to **null**.

8.1.6 Classes Versus Structs

You've seen that a class is a blueprint for objects. Also, variables of a primitive type are passed by value, whereas variables of a class type are passed by reference.

But next to a **class**, C# also knows another object type, called a **struct**. A struct is almost the same as a class: it's a "wrapper" for data (described in member variables) and behavior (described in methods and properties), and you can create multiple instances of it. But in contrast to classes, structs are *directly* stored in memory, just like the primitive types. They are also passed by *value* (and not by reference). In fact, C# uses the term "value type" for primitive types and structs combined.

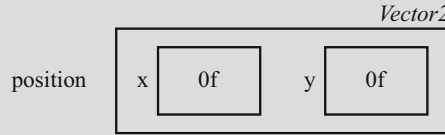
Structs are generally used for basic kinds of objects without too much special behavior. You've already used structs before in this book: the `Vector2` and `Color` types (of the MonoGame engine) are actually structs!

Structs behave like primitive types in several ways. For instance, a variable of a struct type *cannot* be **null**. After all, the keyword **null** indicates a "reference to nothing," and struct variables don't store references at all, but immediate values. Also, if you pass a `Vector2` or `Color` value to a method and then change it inside that method, you're actually changing a *copy* of the value.

The differences between classes and structs are small but important. If your program doesn't do what you expect, it might be due to this difference: objects are being shared by different variables while you didn't count on it or vice versa.

Structs and Default Values — Let's say that a class `MyClass` has a member variable `Vector2` position. If the constructor of `MyClass` does not give position an actual value, then the program will give it a *default value*, just like it does with primitive types such as `float` and `int`.

The default value of a struct is not `null`, because that's not possible for structs. Instead, the program will create a new `Vector2` object with all of its member variables (in this case `float x` and `float y`) set to their default values. So, in the end, the memory will look like this:



8.1.7 Complex Objects

Generally, classes can be quite complicated when represented in memory. An instance of a class can contain variables of other class types, struct types, or primitive types. In turn, any of these types may contain other variables, and so on.

Figure 8.1 shows an impression of what the memory could look like when you create a new instance of a `Cannon` object. You can see the different kinds of objects that are in a `Cannon` instance. The `Vector2` objects are structs, and therefore they are directly part of the `Cannon` instance. `Texture2D` is a class, so the `Cannon` instance contains *references* to `Texture2D` objects.

When you're writing classes of your own, it can sometimes be helpful to draw these kinds of diagrams. This will give you a feeling of which object belongs where and at which places references to these objects are being stored.

8.2 The Ball Class and Its Interaction with the Game World

Now that you know about the representation of objects in memory, it's time to start adding the other objects of the `Painter` game: the ball and the paint cans. This will lead to two new classes `Ball` and `PaintCan`, which both turn out to be quite similar to the `Cannon` class you already have. Both of these classes will be responsible for their own behavior: initialization, input handling, updating, and drawing.

Let's create the `Ball` class first. This section will be long because it involves quite a lot of things: not only the behavior of the ball itself but also a mechanism for giving objects access to each other.

8.2.1 Structure of the Ball Class

Add a new `Ball` class to the project. This class will have many similarities to the `cannon`: exactly the same methods and many of the same member variables and properties.

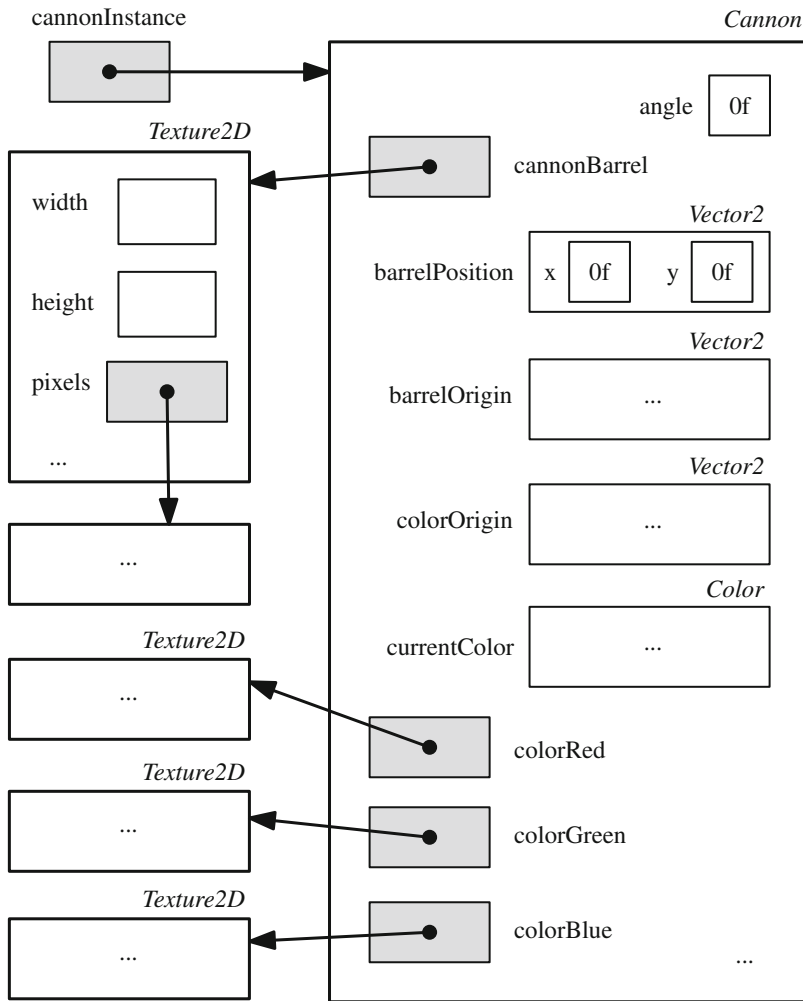


Fig. 8.1 An impression of the memory structure of a Cannon instance

We'll now give an overview of what the class should do. Try to follow this text while you create the *Ball* class yourself. If you get stuck, take a look at the *Painter5a* example project, but be aware that that project already contains spoilers for the rest of this section.

In terms of data, the ball has a position on the screen, an origin for drawing the sprite at the correct place, three differently colored sprites, and a *Color* indicating the current color. Thus, the *Ball* class

should start with the following member variables: *Texture2D colorRed, colorGreen, colorBlue;* You can also
Vector2 position, origin;
Color color;

add the same *Position* and *Color* properties that *Cannon* already has. When copying these properties, be aware that the member variables of *Ball* have slightly different names than those of *Cannon*, so you'll have to change the properties a bit.

Add a constructor method that loads the sprites (don't forget to include them as assets via the Pipeline Tool!), calculates the ball's origin (which is the center point of any of the three sprites), and calls the Reset method to do some final initialization:

```
public Ball(ContentManager Content)
{
    colorRed = Content.Load<Texture2D>("spr_ball_red");
    colorGreen = Content.Load<Texture2D>("spr_ball_green");
    colorBlue = Content.Load<Texture2D>("spr_ball_blue");
    origin = new Vector2(colorRed.Width / 2.0f, colorRed.Height / 2.0f);
    Reset();
}
```

The Reset method should set the ball's initial position and color, because these are the values that can change during the game. For now, we're choosing a starting position so that the ball will be hidden behind the cannon.

```
public void Reset()
{
    position = new Vector2(65, 390);
    Color = Color.Blue;
}
```

Next up, give your class HandleInput and Update methods just like Cannon, but leave these methods empty for now. We'll fill them in later.

Finally, add a Draw method that chooses the appropriate sprite (based on the Color property) and then draws it at the ball's current position:

```
public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    // determine the sprite based on the current color
    Texture2D currentSprite;
    if (Color == Color.Red)
        currentSprite = colorRed;
    else if (Color == Color.Green)
        currentSprite = colorGreen;
    else
        currentSprite = colorBlue;

    // draw that sprite
    spriteBatch.Draw(currentSprite, position, null, Color.White,
        0f, origin, 1.0f, SpriteEffects.None, 0);
}
```

This concludes the Ball class for now. To add an instance of this class to the game world, give the GameWorld class another member variable:

```
Ball ball;
```

and initialize it in the constructor of GameWorld:

```
ball = new Ball(Content);
```

Also, call ball.HandleInput and ball.Update inside the appropriate methods of GameWorld. Finally, in the Draw method of GameWorld, you should draw the ball just *before* you draw the cannon. That way, it will look as if the ball is inside the cannon.

If you compile and run the game now, you will still see the same thing as before. This is because you're currently drawing the ball behind the cannon. (To check that the ball really does exist, you could temporarily comment out the line that draws the cannon and run the game again.) We still have to give the ball a correct position on the screen.

8.2.2 Overview of the Ball Behavior

What should a Ball object do in the game? Initially, it should be “stuck” at the tip of the cannon barrel, waiting to be launched. Until it launches, it should have the same color as the cannon. If the player clicks the left mouse button, we want the ball to fly away in the direction that the cannon is pointing to. At that point, the color of the ball can’t change anymore. The player can shoot only one ball at a time. Later, if the ball touches a paint can, this should affect the paint can, and a new ball should appear inside the cannon.

There are many ways to implement all of this behavior. We’ve made the following design choice: there is always *one* ball object in the game, and it’s part of the game world right from the start. The ball has a member variable that says whether or not it is currently flying through the air. If it is, it will update its position and check for collisions. If it’s not, it will update its color (to match the color of the cannon), and check if the user has clicked the left mouse button (so that it can start flying).

The first challenge is to draw the ball at the right position (the tip of the cannon barrel) and in the right color (the same color as the cannon). You will do this by updating the ball’s color and position member variables inside the Update method. But for both of these updates, you need to ask the *cannon* for details!

How can we make that possible? As usual, there are many possible options, and there’s not really a best or worst one. You could do any of the following things:

- Give the ball a *reference* to the cannon, using a method parameter or a member variable.
- Let the *cannon* be in charge of updating the ball, instead of the ball itself.
- Let the *game world* be in charge of updating the ball.
- Make the cannon available via the game world, so that the ball can use it when needed.

We’ve chosen to implement the last option. This is also a good excuse to teach you another new programming concept: the keyword **static**.

8.2.3 *static*: Making the Game World Accessible Everywhere

In short, we’ll create a way to make the cannon’s data available *everywhere* in the program, via the GameWorld instance that stores it. Remember: the cannon is currently stored in the cannon member variable of the game world. And in turn, this game world is stored in the gameWorld member variable of the overall Painter class. To make sure that the ball can retrieve the cannon inside the game world, you need to do two things: expose the gameWorld variable of the Painter class and expose the cannon variable of the GameWorld class.

Let’s create two *read-only properties* for this. Add the following property to the GameWorld class:

```
public Cannon Cannon
{
    get { return cannon; }
}
```

and add the following property to the Painter class:

```
public GameWorld GameWorld
{
    get { return gameWorld; }
}
```

Do you see the point here? Inside the main Painter class, you can now access the cannon via GameWorld.Cannon, using both properties combined.

You're almost ready to use the cannon from within the `Ball` class now, but there's still one missing link. When you want to call a method or use a property of a class, you need a specific *instance* of that class. You've seen this in instructions like `spriteBatch.Draw` (where `spriteBatch` is a particular instance of the class `SpriteBatch`) and `cannon.HandleInput` (where `cannon` is a particular instance of the class `Cannon`).

This also means that you would need a specific instance of the `Painter` class before you can use the `GameWorld` property. To retrieve the cannon of the game world anywhere in your code, you'd have to write something like this:

```
painterInstance.GameWorld.Cannon
```

where `painterInstance` somehow stores an instance of `Painter`. But do we really want to give each game object its own reference to the overall `Painter` object? That would make our program pretty chaotic. Luckily, there's another solution: the keyword **static**.

You can add the keyword **static** in front of a method, a property, or a member variable. If you do this, it will mean that the method, property, or member variable gets *shared by all instances* of a class. This way, if you want to use that method/property/variable, you don't need a specific *instance* of the class but only the *name* of the class.

In this game, we know that there's always only a single game world. So, in the `Painter` class, you can turn the `GameWorld` property into a **static property**, by simply adding the keyword **static** to its header:

```
public static GameWorld GameWorld
{
    get { return gameWorld; }
}
```

To make this code work, you'll also have to make the `gameWorld` member variable static:

```
static GameWorld gameWorld;
```

(We'll explain why soon.) You can now write `Painter.GameWorld.Cannon` to retrieve the cannon of the game world anywhere in your code. As you can see, instead of an *instance* of the `Painter` class, you now only have to write the *name* of the `Painter` class. This is enough because the `GameWorld` property gets shared by all instances of `Painter`, so you don't need to specify a particular instance anymore.

You could say that a static method (or property, or member variable) affects a class as a whole, whereas a non-static method affects only one instance of a class.

The keyword **static** finally explains why (in the code examples you've seen so far) you sometimes needed to write a *variable name* in front of a dot (such as `spriteBatch.Draw`) and sometimes a *class name* (such as `Mouse.GetState`). Whenever you wrote the class name instead of an instance name, you were actually using something *static*—we just hadn't told you yet. Perhaps without knowing it, you've been using several static methods and properties already:

- `Color.Red` is a *static property* of the `Color` struct. It creates and returns a new `Color` object, representing the color red. The creators of `MonoGame` made this property static, because you don't need a specific `Color` instance yet if you want to create a new color.
- Similarly, `Vector2.Zero` is a *static property* of the `Vector2` class. It gives you a new `Vector2` object (with both coordinates set to zero), but you don't need a specific `Vector2` instance to retrieve it.
- **static void** `Main()` is the default starting point of every C# program. It's a *static method*, and for good reason. When the program starts, not a single object has been created yet, so you cannot possibly call a method that belongs to a specific object. Therefore, the very first method of your program *must* be static.
- `Mouse.GetState` and `Keyboard.GetState` are *static methods* of the classes `Mouse` and `Keyboard` in `MonoGame`. You can call these methods from anywhere, without requiring a specific instance of a `Mouse` or `Keyboard`. The creators of `MonoGame` have done this to give you easy access to the mouse and keyboard, anywhere in your code.

One last note: if you're inside a static method or property, you cannot do anything that is *not* shared by all instances of your class. For example, you cannot retrieve the value of a non-static member variable: each class instance has its own version of that variable, but you're currently not dealing with any specific instance, so the compiler can't possibly know what you mean. That's why you had to make the `gameWorld` variable static as well: otherwise, our static `GameWorld` property would try to use something non-static.

For the same reason, you can't use the keyword **this** in a static method: there is no "current instance" to refer to!



Quick Reference: The Keyword **static**

If you mark a method (or a property) as **static**, then it gets shared by all instances of its class. To call the method from outside that class, you no longer need a specific instance of the class. Instead, you only need the class *name*.

For instance, if a class `MyClass` has a method with the following header:

```
public static void DoSomething()
```

then you can call it elsewhere by writing `MyClass.DoSomething()`.

You can also create static *member variables*. A static member variable also gets shared by all class instances, so it will have a single value for all instances combined.

Inside a static context, you cannot do anything that *does* require a specific instance of your class. For example, you cannot use non-static member variables or the keyword **this**.

By the way, there are good reasons for using the keyword **static** as little as possible.

Don't Use **static too Often** — Public static methods and properties are useful for making certain things available everywhere in your program. In this case, the static `GameWorld` property makes sure that every object in the game can obtain details about the game world.

This raises a question: if the keyword **static** magically solves our problems, why don't we just mark *all* variables, methods, and properties as static (and public)? You'd be able to access everything everywhere! That may sound ideal, but it's not a good idea.

First of all, making everything static creates the impression that you'll only ever need a single instance of everything. As a result, you're basically ignoring half of the power that object-oriented programming offers.

Second: this may sound weird, but it's actually a *good* thing that objects are not easily accessible. This forces programmers to put their code in logical places. For instance, the `inputHelper` object isn't static and is passed along as a parameter to the `HandleInput` methods. This way, input handling *has to be done* in the `HandleInput` method and not elsewhere, because that's the only place where there is access to the `inputHelper` object. Similarly, the `spriteBatch` object is only available in the `Draw` method. If there would be a static way to get these objects, then you could go totally crazy: you could handle input in the `Draw` method or try to draw sprites in the `HandleInput` method. This would completely mess up the design of your program.

(continued)

In conclusion: the keyword **static** is useful if there's only a single instance of an object *and* if it makes sense to use that object in many different places. In all other cases, you should avoid this keyword, because (to some extent) it beats the purpose of object-oriented programming.

8.2.4 Drawing the Ball at the Cannon Tip

Thanks to the static `GameWorld` property, you can now finally fill in the `Update` method of the `Ball` class. The following instruction lets the ball's color match the cannon's color:

```
Color = Painter.GameWorld.Cannon.Color;
```

The second task of `Update` is to give the ball the correct position, exactly at the tip of the cannon barrel. Let's give the `Cannon` class a (read-only) property for this. It computes and returns a new `Vector2` object, representing the point on the screen that lies exactly on the cannon's tip. This position depends on the angle variable, of course. Here's the code that gets the job done:

```
public Vector2 BallPosition
{
    get
    {
        float opposite = (float)Math.Sin(angle) * cannonBarrel.Width * 0.75f;
        float adjacent = (float)Math.Cos(angle) * cannonBarrel.Width * 0.75f;
        return barrelPosition + new Vector2(adjacent, opposite);
    }
}
```

Because this is a public property, you can use it in the `Update` method of `Ball`, via the same (static) expression `Painter.GameWorld.Cannon`:

```
position = Painter.GameWorld.Cannon.BallPosition;
```

This sets the position of the ball to the position that the `Cannon` instance computes for us.



CHECKPOINT: Painter5a

If you compile and run the game now, you'll see the ball being drawn at the tip of the cannon barrel. Also, if you press the R, G, or B key, the ball's color will change along with the cannon's color, because the ball matches its color with the cannon in every frame.

Nice work! You've created a game world with multiple game objects, and you've developed a nice way to make these objects use each other's data. With this structure in place, it's time to spice up the behavior of the ball.

8.2.5 Letting the Ball Move

To keep track of whether or not the ball is currently flying around, give the `Ball` class an extra member variable:

```
bool shooting;
```

If this variable is **false**, the ball will be waiting inside the cannon. If the variable is **true**, the ball will be flying through the screen.

When the ball is flying, we want to update its position according to a certain *velocity*. For this, add another member variable:

```
Vector2 velocity;
```

You could see the velocity variable as the number of pixels that the ball will move per second. The *x*-coordinate of the velocity vector stores the horizontal speed (in pixels per second), and the *y*-coordinate stores the vertical speed.

In the Reset method of Ball (which also gets called in the constructor), set these two new variables to **false** (not flying) and Vector2.Zero (no movement):

```
public void Reset()
{
    position = new Vector2(65, 390);
    velocity = Vector2.Zero;
    shooting = false;
    Color = Color.Blue;
}
```

Next, let's fill in the HandleInput method of Ball. If the player presses the left mouse button, we want to set the shooting variable to **true**, and we want to give the ball a nice velocity. For now, let's move the ball to the right in a straight line.

Also, the player should be allowed to launch the ball only once. As soon as the ball is flying, you shouldn't be allowed to launch it again. This means that the player should only be allowed to shoot the ball when the shooting variable is currently **false**.

The following instructions do exactly what we want:

```
if (inputHelper.MouseLeftButtonPressed() && !shooting)
{
    shooting = true;
    velocity = new Vector2(100,0);
}
```

(The chosen value of velocity will make sure that the ball moves to the right, at a speed of 100 pixels per second.) Note: `inputHelper.MouseLeftButtonPressed()` is a Boolean expression that is **true** when the player has just pressed the left mouse button. And `!shooting` is a Boolean expression that contains the logical “not” operator: it is **true** exactly if shooting itself is **false** and vice versa. So these two expressions combined (with the logical operator `&&` in-between) form the if-condition you're looking for.

In the Update method, you want to give the ball two different types of behavior, depending on the shooting variable. Fill in this method as follows:

```
public void Update(GameTime gameTime)
{
    if (shooting)
    {
        position += velocity * (float)gameTime.ElapsedGameTime.TotalSeconds;
    }
    else
    {
        Color = Painter.GameWorld.Cannon.Color;
        position = Painter.GameWorld.Cannon.BallPosition;
    }
}
```

What does this method do? The **if** block updates the ball's position, if the shooting variable is currently storing the value **true**. The velocity code is a bit complicated (but very important!), so we'll get back to it soon.

The **else** block contains the ball's behavior for when it is still inside the cannon. These are same two instructions that you had before. These instructions will be executed whenever the shooting variable does *not* store the value **true** (in other words, when it is **false**).

Now, take a closer look at the instruction that changes the ball's position:

```
position += velocity * (float)gameTime.ElapsedGameTime.TotalSeconds;
```

We said before that the **velocity** variable stores the ball's movement speed in pixels per second. But each *frame* of the game loop is much shorter than a second. If you'd simply write `position += velocity;`, you would move the ball way too quickly: it would receive a full second worth of movement within just one frame! To compensate for that, you need to multiply the velocity by the number of seconds that has passed in the current frame. The expression `(float)gameTime.ElapsedGameTime.TotalSeconds` gives you this number.

Just to be complete: `ElapsedGameTime` is a (non-static!) property of the `GameTime` class, so `gameTime.ElapsedGameTime` gives you a result that is based on the `gameTime` parameter. This result has the type `TimeSpan`, which contains another property called `TotalSeconds`. That property gives you the total number of seconds (as a fractional number) that has passed since the last frame. So, `gameTime.ElapsedGameTime.TotalSeconds` gives you the time that has passed since the last frame, in seconds. However, this expression has the data type **double**, and you're only allowed to multiply `Vector2` objects by a value of type **float**. Therefore, you have to *cast* the expression to a **float** before you can use it in the multiplication.

In the games throughout this book, many game objects will have a position and a velocity that work like this. In their `Update` method, these objects will update their position by adding their velocity to it, multiplied by the time that has passed in a frame.

Multiplying by the Time — There's another reason why it's very important to adapt the velocity to the time that has passed. Assume that you *don't* do this. If your game ever gets a lower framerate because the computer is too busy, then everything will move more slowly! The opposite is also true: if the framerate ever *increases* (e.g., because you allow the game to run at more than 60 FPS), your objects will become faster!

This sounds like a silly mistake to make, but it actually happened in some very old games, designed for very old computers (when the thought of 60 FPS was still science fiction). As computers became faster, these games could be played at a higher framerate, up to the point where they became unplayable because everything was moving way too fast.

Let this mistake from the early days be a lesson: if you express an object's velocity in *pixels per second* (and not in *pixels per frame*), the game's behavior will become independent of the framerate, which is much nicer.

Compile and run the game again. As soon as you've pressed the left mouse button, the R/G/B keys will no longer change the color of the ball. The ball will then also start moving to the right, until it leaves the screen and you can't see it anymore.

Okay, so we can launch the ball now, but that's about it. Let's add three more features:

- The launch velocity of the ball should depend on the mouse position;
- The ball should experience gravity;
- The ball should return to its "nonflying" state when it leaves the screen.

8.2.6 Giving the Ball the Correct Launch Velocity

For the ball's launch velocity, imagine that you draw a line from the cannon's center to the mouse position. We'd like our ball to move along this line. In other words, this line indicates the desired direction of the ball. The line is given by the following expression:

```
inputHelper.MousePosition - Painter.GameWorld.Cannon.Position
```

This uses the `-` operator, which MonoGame has implemented for the `Vector2` data type. It computes the *difference* between two vectors and returns that difference as a new `Vector2` object. If you imagine that two vectors A and B are positions in the game world, then you could interpret `B - A` as the "arrow" that goes from A to B.

As you can see, we use the same `Vector2` type for many different things: a position on the screen, a movement speed, or a line from one position to another. This is all perfectly fine: in the end, a vector is just an object with an *x*- and *y*-coordinate, so it can describe anything that uses only those two components.

Next to a *direction*, the expression above also has a *magnitude*, which you could interpret as the length of the arrow. If the mouse is farther away from the cannon, the calculated vector will have larger *x* and *y* components. The effect will be that the ball moves faster if the mouse is farther away from the cannon, which is exactly what we want.

Inside the `HandleInput` method, change the line that sets the velocity, like this:

```
velocity = (inputHelper.MousePosition - Painter.GameWorld.Cannon.Position) * 1.2f;
```

Do you see the same "`B - A`" construction here? We've also multiplied that vector by a number (which is allowed in MonoGame), to make it a bit bigger—in other words, to let the ball move a bit faster.

Compile and run the game again. You can now shoot the ball into the direction indicated by the cannon barrel. Also, if you hold the mouse pointer farther away from the cannon, you'll launch the ball at a higher speed. (But still, you can do this only once per game, because the ball never gets reset to its initial state.)

Gameplay Parameters — We've chosen the value of `1.2f` after a couple of play sessions. Each game will have a number of these *gameplay parameters* that will need to be tweaked while play-testing the game, to determine their optimal value. Finding the right values for these parameters is crucial for a balanced game that plays well. You need to make sure that the chosen values do not make the game too easy or difficult. For example, if the ball in Painter moves too slowly, it would make the game almost unplayable because you cannot quickly shoot a new ball.

8.2.7 Applying Gravity

In real-world physics, gravity is a force that pulls everything down. As long as an object is falling, its vertical speed will keep increasing because of that force. So, gravity is an *acceleration* factor that increases an object's vertical speed over time.

To translate this to the ball in Painter, we want to slightly increase the ball's vertical speed in every frame of the game loop. In other words, the `Update` method should add a little bit to the *y* component of the velocity vector. And this "little bit" should (again) depend on the number of seconds that has passed

per frame; otherwise the ball's behavior will be affected by the game's framerate. The following line does the job:

```
velocity.Y += 400.0f * (float)gameTime.ElapsedGameTime.TotalSeconds;
```

We've chosen the constant 400.0f via trial and error again. Whether it's realistic or not, it gives the ball a nice-looking gravity effect.

Game Physics — In the real world, the gravity force is not 400. But then again, the real world doesn't consist of pixels either. When you want to incorporate physics in your game, the most important thing is not that the physics are realistic, but that the *game is playable*. Your choices have a huge influence on the gameplay. Try to give the physics parameters of Painter different values, just to see what happens. You'll see soon enough that there's only a small range of values that gives you a nicely playable game.

You're now writing `(float)gameTime.ElapsedGameTime.TotalSeconds` two times in the same method. At this point, it makes sense to create a *helper variable* that stores this value, so that you don't have to calculate it twice. The code inside your `if` block will then look like this:

```
float dt = (float)gameTime.ElapsedGameTime.TotalSeconds;
velocity.Y += 400.0f * dt;
position += velocity * dt;
```

8.2.8 Resetting the Ball When It Leaves the Screen

Finally, we'd like to reset the ball when it has moved outside the screen. You can reset the ball by calling the `Reset` method: this will set the variable shooting back to **false**, and it will set the velocity back to zero.

An easy (but not yet perfect) solution is to reset the ball whenever it has moved too far to the right. Add the following lines to the `Update` method, right after the code that updates the ball's position:

```
if (position.X > 800)
    Reset();
```

This will reset the ball as soon as it is more than 800 pixels to the right of the screen's left boundary. This is almost good enough, but not completely. For a perfect solution, the ball should find out how large the screen is, instead of relying on a "hard-coded" value of 800. That way, if we ever make the screen wider, the code will still work.

To allow that, let's add another static property to the `Painter` class. This property will give us a `Vector2` object that stores the width and height of the screen, in pixels.

First, give the `Painter` class a static property:

```
public static Vector2 ScreenSize { get; private set; }
```

The **set** part of this property is private, so that only `Painter` itself can fill it in. The **get** part is public so that all classes can access it. (Also, this is one of those "shorthand" properties: its **get** and **set** parts have no body, and there's no explicit member variable behind it.)

Set this property inside the `LoadContent` method:

```
ScreenSize = new Vector2(GraphicsDevice.Viewport.Width, GraphicsDevice.Viewport.Height);
```

The expression `GraphicsDevice.Viewport.Width` gives you the width of the screen. This is a value of type **int**, so you can safely pass it to the constructor of a `Vector2` object.

Now, the expressions `Painter.ScreenSize.X` and `Painter.ScreenSize.Y` will give you the width and height of the screen, anywhere in the program.

The idea is to reset the ball when it has passed the right side of the screen, or the left side of the screen, or the bottom of the screen. We *do* allow the ball to move above the top of the screen. This allows the player to shoot a ball high into the air. You often see this effect in platform games as well: a character can often jump “out of view” at the top without dying.

Later, you’re going to use the same mechanism to also reset the *paint cans* when they’ve left the screen. For that reason, it’s good to already create a method for this, so that all game objects can use it. This avoids duplicate code later on.

We think it makes the most sense to put this method in the `GameWorld` class, because the game world is as large as the screen. Therefore, you could say that this method tells us if a point lies inside or outside the game world. Add the following method to `GameWorld`:

```
public bool IsOutsideWorld(Vector2 position)
{
    return position.X < 0 || position.X > Painter.ScreenSize.X
        || position.Y > Painter.ScreenSize.Y;
}
```

This method calculates a Boolean expression with three parts, connected by logical “or” operators (`||`). This expression is **true** as soon as *at least one* of the three parts is true. So, this expression tells us whether a point lies to the left of the screen, to the right of the screen, or below the screen.

You can now use this new method to properly check whether the ball has left the game world. In the `Update` method of `Ball`, replace the last `if` block by the following:

```
if (Painter.GameWorld.IsOutsideWorld(position))
    Reset();
```



CHECKPOINT: Painter5b

And that’s it! Compile and run the game, and you’ll have a fully functioning ball that automatically returns to the cannon when it has left the screen.

8.3 The PaintCan Class

The Painter game should contain three paint cans that fall down. In this section, you will add those paint cans to the game. You’ll create a new `PaintCan` class for this. The three paint cans will be three different *instances* of the same class. This shows that a class can really describe multiple game objects at the same time, each with their own position, color, and so on. In other words, `PaintCan` will describe what a paint can is and what it can do, and it’s a blueprint that you can use to create multiple paint cans.

The results for this section can be found in the `Painter6a` and `Painter6b` programs.

8.3.1 Structure of the Class

Start by including three new sprites in your project: `spr_can_red.png` and its green and blue variants. Then create a new class `PaintCan`, in its own new file. This class will have many similarities to `Ball` and `Cannon`.

A paint can stores (at least) a position, an origin, a velocity, three colored sprites, and the current color. These are the exact same member variables that Ball also had, except for the variable **bool** shooting, which we don't need here. Go ahead and copy these member variables from Ball. You can also copy the Draw method, the Position property, and the Color property while you're at it.

Also, add a constructor that loads the appropriate sprites, sets the origin to the center of a sprite, and calls the Reset method. That method should set the color to blue and the velocity to zero. Don't worry about the *position* of the paint can yet: we'll handle that soon.

The paint cans don't need input handling, but they do need to move in every frame. So, you don't need a HandleInput method, but you do need an Update method. Feel free to copy this method from the Ball class and then remove its contents.

Listing 8.1 shows the PaintCan class so far.

8.3.2 Creating Three Different Paint Cans

The GameWorld class should store three instances of PaintCan, in member variables:

Listing 8.1 The basis of the PaintCan class

```

1  using Microsoft.Xna.Framework;
2  using Microsoft.Xna.Framework.Graphics;
3  using Microsoft.Xna.Framework.Content;
4
5  class PaintCan
6  {
7      Texture2D colorRed, colorGreen, colorBlue;
8      Vector2 position, origin, velocity;
9      Color color;
10
11     public PaintCan(ContentManager Content)
12     {
13         colorRed = Content.Load<Texture2D>("spr_can.red");
14         colorGreen = Content.Load<Texture2D>("spr_can.green");
15         colorBlue = Content.Load<Texture2D>("spr_can.blue");
16         origin = new Vector2(colorRed.Width, colorRed.Height) / 2;
17         Reset();
18     }
19
20     public void Update(GameTime gameTime)
21     {
22         // TODO: We'll fill in this method soon.
23     }
24
25     public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
26     {
27         // same as Ball
28     }
29
30     public void Reset()
31     {
32         color = Color.Blue;
33         velocity = Vector2.Zero;
34     }
35
36     // Position and Color properties: same as Ball
37 }
```

```
PaintCan can1, can2, can3;
```

We'll use the same trick that we used for the ball: the `PaintCan` instances are always part of the game world, and we just reset their positions when they leave the screen. That way, you don't have to destroy any of the instances (or create new ones) during the game. You can initialize these variables in the constructor of `GameWorld`, as follows:

```
can1 = new PaintCan(Content);
can2 = new PaintCan(Content);
can3 = new PaintCan(Content);
```

But this will create three objects with the exact same starting situation. What we want is that each paint can starts at a different position (a different x -coordinate). Also, remember the final goal of the game: the player should shoot balls to change the colors of paint cans, so that they receive the correct color before they leave the screen. This means that every paint can should have its own *target color*: the color that the can wants to have by the time it leaves the screen. The first paint can has the target color red, the second green, and the third blue.

To describe these differences between the paint cans, add two extra parameters to the `PaintCan` constructor:

```
public PaintCan(ContentManager Content, float positionOffset, Color target)
```

The `target` parameter indicates the can's target color. This is something that the paint can needs to remember during the game, so you need another member variable for it:

```
Color targetcolor;
```

And in the constructor, store the `target` parameter inside the `targetcolor` member variable:

```
targetcolor = target;
```

The constructor's `positionOffset` parameter indicates at what x -coordinate the can should be drawn. This is something that you can immediately store in the `position` member variable. You don't need an extra member variable for it, because the x -coordinate of the can will not change during the game. Go ahead and initialize the `position` inside the constructor:

```
position = new Vector2(positionOffset, 100);
```

And finally, because we've added two new parameters to the constructor, we now also need to *specify* these parameters when `GameWorld` creates the actual `PaintCan` instances. Change the initialization of `can1`, `can2`, and `can3` into this:

```
can1 = new PaintCan(Content, 480.0f, Color.Red);
can2 = new PaintCan(Content, 610.0f, Color.Green);
can3 = new PaintCan(Content, 740.0f, Color.Blue);
```

This creates the `PaintCan` objects correctly, each with different details.

Finally, in the `Update` and `Draw` methods of `GameWorld`, don't forget to call the `Update` and `Draw` methods of all three paint cans. Otherwise, the cans will never be updated or shown! If you compile and run the game after that, you will see three paint cans in the game, at different x -coordinates. All three cans will be blue, because that's the color they all receive in the `Reset` method. They do have different *target* colors (namely, red, green, and blue), but you can't see that yet.



CHECKPOINT: Painter6a

8.3.3 Making the Paint Cans Loop Around

The next step is to let the paint cans move down and to revert them to the starting position when they've left the screen. First, in the `PaintCan` constructor, change the starting position so that the paint can is just outside the screen:

```
position = new Vector2(positionOffset, -origin.Y);
```

(Make sure to do this *after* you've calculated the origin.) Next up, in the `Reset` method, you'll want to reset the can's *y*-coordinate so that the object moves back to the top of the screen:

```
position.Y = -origin.Y;
```

We also want to make the paint cans go faster during the game. We do this by defining a *minimum speed* that increases over time. Give `PaintCan` a new member variable for this:

```
float minSpeed;
```

And in the constructor of `PaintCan`, set this to an initial value:

```
minSpeed = 30;
```

Now let's fill in the `Update` method. This method should first increase the `minSpeed` value according to the time that has passed. With the following code, the speed will get 0.01 larger for each second that passes:

```
float dt = (float)gameTime.ElapsedGameTime.TotalSeconds;
minSpeed += 0.01f * dt;
```

Next, if the paint can is currently *not* standing still, it should move according to its velocity (just like in the `Ball` class). Also, when the can has left the screen, the object should reset itself (again: just like `Ball`). You can do this as follows:

```
if (velocity != Vector2.Zero)
{
    position += velocity * dt;
    if (Painter.GameWorld.IsOutsideWorld(position))
        Reset();
}
```

Below that `if` instruction, you can add an `else` block. This code will be executed when the paint can is *not* moving. This will happen at the start of the game and every time that the paint can has just reset itself. In this `else` block, you need to give the can a velocity that lets it move straight down, using the `minSpeed` variable. The following code does this:

```
else
{
    velocity = new Vector2(0, minSpeed);
}
```

Run the game again and you'll see that the paint cans are indeed moving down and resetting themselves when they're near the bottom of the screen. And after every reset, the cans will move a little bit faster—although you'll need to run the game for a while before you can notice that.¹

8.3.4 Taking the Origin into Account

If you look at the game critically, you'll see that the cans disappear a bit too early. This is because a paint can resets itself when its *origin* has left the screen. But at that point, part of the sprite is still

¹Good excuse to take a coffee break while the game is running.

visible. We'd like to change this so that the paint can resets itself when the *top* of its sprite has left the screen.

The position of the paint can's top-left corner is given by this expression:

`position - origin`

Why is that? Well, imagine that the origin of the sprite lies at pixel (100, 100) of that sprite. If you draw the sprite at a position (x, y) with that origin, then MonoGame will actually draw the entire sprite 100 pixels to the left of that point and 100 pixels above that point. This makes sure that the *origin* of the sprite lies exactly at position (x, y) . In other words, the top-left corner will actually be drawn at $(x - 100, y - 100)$. You can also write that as $(x, y) - (100, 100)$.

So, in general, to determine where the top-left corner of the sprite will be drawn, you need to subtract the origin vector from the position vector. This is exactly given by the expression `position - origin`.

To make sure that the paint can gets reset when the *entire* sprite has left the screen, change the "reset" part of Update method into this:

```
if (Painter.GameWorld.IsOutsideWorld(position - origin))
    Reset();
```

This fixes the problem and gives you a nicer-looking game.

8.3.5 Giving the Cans Random Speeds

But now, all paint cans are moving at exactly the same speed. It'd be nice to make the game less predictable, by adding a bit of *randomness* to the speed of the cans. Of course, this randomness factor shouldn't be too big: we don't want one can to take 3 h to fall from top to bottom while another can takes only 1 ms. The speed should be random, but within a *playable range* of speeds.

What does randomness actually mean? In the real world, a nice example of randomness is throwing a die: this gives you a truly random integer between 1 and 6. Similarly, flipping a coin gives you a result of either *heads* or *tails*, which you could see as a random integer between 0 and 1. We'd like to do something like this in our game, to add a random factor to the speed of a paint can.

But a computer can't flip coins or roll dice for you. Luckily, programming languages (such as C#) have something that comes very close to that: **random number generators**. A random number generator is a special object in the program that can give you random numbers. Don't worry about how this works exactly: it's a pretty complicated process. Let's just be happy that random number generators exist, and focus on how you can *use* them.

In C#, generating random numbers is done by using the `Random` class from the `System` library. Let's create a *single* instance of this class, inside the `Painter` class. (You'll have to include the `System` library in that class.)

We want to make sure that only the `Painter` class itself can *create* this instance but that other classes can still *access* it. Can you guess how we should do that? Well, the shortest solution is to add a *property* with a public **get** part and a private **set** part, just like what you did earlier for the screen size. Here's what it could look like:

```
public static Random Random { get; private set; }
```

An alternative solution would be to create a private *member variable* and a public *read-only property*, just like what you did for the game world:

```
static Random random;
public static Random Random { get { return random; } }
```

This solution is pretty much equivalent to the previous one. In our example program, we'll go for the first option because it's shorter. Assuming you've made the same choice, don't forget to initialize this new property in the constructor of Painter:

```
Random = new Random();
```

The Random class has two interesting methods that you can use. The first method is called Next: it gives you a random **int** within a certain range that you provide via parameters. All possible integers in the range have an equal chance of being chosen. The second method is called NextDouble: it gives you a random **double** between 0 and 1, so this can be 0.0052 or 0.4815162342 or anything else. For instance, if you want to create a random **double** between 0 and 500, you call the NextDouble method and multiply the result by 500. Likewise, if you want a random number between 0 and $\frac{1}{2}$, you call the NextDouble method and divide the result by 2.

Let's put this to good use by giving each paint can a random velocity. Give the PaintCan class the following extra method:

```
Vector2 CalculateRandomVelocity()
{
    return new Vector2(0.0f, (float)Painter.Random.NextDouble() * 30 + minSpeed);
}
```

What does this method do? It first uses the static Painter.Random property to get the game's random number generator. Then it calls the NextDouble method for that object, which gives us a random **double** value between 0 and 1. This result is then converted to a **float**. Next, that number is multiplied by 30, so we get a random **float** value between 0 and 30. This number is added to the minSpeed value that PaintCan was already using.

In the end, this method creates a vertical speed vector that always has a *y* component of *at least* minSpeed, but the actual value will be between 0 and 30 larger than that. We don't exactly know *how much* larger it will be: the result will be different every time you call the CalculateRandomVelocity method.

The final step is to actually *use* this method somewhere. Go to the Update method of PaintCan, and look for the line that currently sets the velocity to new Vector2(0, minSpeed). Replace it with this line:

```
velocity = CalculateRandomVelocity();
```

This sets the velocity to a new random value, which will be recalculated each time (via the new method that you've just created). Note that the velocity is only updated when the can *starts falling*. During a single fall, the velocity doesn't change.

If you compile and run the game now, the three paint cans will fall down at different speeds. The speed of a can will be recalculated every time the can is reset. So the left can might move fast on its first trip, then slow on its second trip, and so on. But on the long term, the cans will become gradually faster, thanks to the minSpeed variable that builds up over time.

Random Number Generators — Are you curious about *how* a computer creates random numbers? Well, that's a long story, but we'll try to give you a short explanation. Feel free to stop reading if you get dizzy!

The numbers that a computer program creates are never *truly* random. Instead, they are the result of a complicated mathematical process. This process takes a few parameters to start with, and it can then calculate a *sequence* of numbers that *seem* random to our human eyes, even

(continued)

though they really aren't. Every time you call the `Next` or `NextDouble` method of a `System.Random` object, you'll use the next number in that mysterious sequence.

If you give two `Random` objects the exact same parameters to start with, they will both give you the exact same sequence of numbers. But if you give them *different* starting parameters, they will (most likely) give different numbers. In C# (and many other languages), random number generators have a single starting parameter, which is called the **seed**. This seed determines exactly which numbers get drawn.

When creating a new `Random` object, you can give that object your own seed value (if you want to). There's a second version of the `Random` constructor that takes an `int` parameter, which represents the seed. However, if you *don't* specify your own seed, then the program will automatically choose a seed for you. This happens in the `Random` constructor with zero parameters. In that constructor, the computer picks a seed that will be different each time, such as the current time stored on the computer. As a result, the random number generator will generate different numbers each time, so your program will behave differently every time you start the game.

Officially, the generated numbers are actually called **pseudo-random**. If you know the seed of a random number generator, the generated numbers are completely predictable. (By contrast, the outcome of a coin flip is *not* predictable, so that's a *true* random process.)

8.3.6 More Randomness

Let's finish this section by adding two more types of randomness to the `PaintCan` class.

First, we'd like to give a can a *random color* every time it starts falling. This will make the game more interesting to play. It will also create the illusion that you're seeing a different can each time (instead of the same can that's recycling itself).

For calculating a random color, we also use the random number generator, but we use the `Next` method instead of the `NextDouble` method. This method is useful if you want a random *integer* or if you want to choose randomly between a fixed number of options. The `Next` method has a parameter that indicates the number of different options you want. For example, the call `Painter.Random.Next(10)` will give a random number between 0 and 9 (note: the value 10 itself will be excluded).

For the paint cans, we want to choose a random color from *three* options: red, green, or blue. So, you can use `Painter.Random.Next(3)` to draw a random number (0, 1, or 2) and then use `if`-instructions to choose a different color in each case. Let's put this in a new method of the `PaintCan` class. This is what the full method looks like:

```
Color CalculateRandomColor()
{
    int randomval = Painter.Random.Next(3);
    if (randomval == 0)
        return Color.Red;
    else if (randomval == 1)
        return Color.Green;
    else
        return Color.Blue;
}
```

As an exercise for yourself, read this method carefully and try to understand completely what it does. The method combines a lot of programming concepts from this book!

You'll want to call this method every time a paint can starts falling again. That's the same place in your code where you're already calculating a random velocity. Find that line (in the `Update` method of `PaintCan`) and add the following line below it, inside the same **else** block:

```
Color = CalculateRandomColor();
```

Compile and run the game again. Every paint can will now receive a new random color each time it starts falling.

There's one more random factor to add: when a paint can has left the screen, we'd like to wait a random amount of time before it starts falling again. To achieve that effect, you can add something to the **else** block of the `Update` method.

Let's apply the following trick: in each frame of the game loop, there will be a 1% chance that the can starts falling. This is pretty easy to do with the `Random` class: you can draw a random **double** value between 0 and 1, and check if that value is smaller than 0.01. If that's true, the program "passes the test," and we'll allow it to set the velocity and color of the can.

So, in the end, you need to add another **if** condition to the **else** block of `Update`. That way, the body of the **else** block will only be executed when this new extra test succeeds. This is what the complete block should look like:

```
else if (Painter.Random.NextDouble() < 0.01)
{
    velocity = CalculateRandomVelocity();
    Color = CalculateRandomColor();
}
```

Compile and run again. As you can see, the paint cans no longer restart immediately after they've left the screen. Instead, they wait a random amount of time.



CHECKPOINT: Painter6b

The `PaintCan` class is now finished. Your game is already starting to look a lot like the full `Painter` game!

8.4 Handling Collisions Between the Ball and Cans

The overall topic of this chapter is "communication and interaction between objects." In light of that topic, there's one more thing we need to add to the `Painter` game: handling collisions between the ball and the paint cans. When the ball touches a paint can, that paint can should receive the same color as the ball, and then the ball should reset itself.

In general, when you want to handle a collision between two (types of) game objects, you can write the code in either of the two classes that are involved. A collision between a `Ball` instance and `PaintCan` instance could be handled in either the `Ball` class or the `PaintCan` class. In this chapter, we'll choose to handle the collision in the `PaintCan` class, because then each paint can instance only has to look at a single ball. If we did it the other way around, we'd have to let the `Ball` class check for three different paint cans. That would lead to duplicate code pretty quickly, and we'd have to change it again if we ever want to add a *fourth* paint can to the game. By handling collisions in the `PaintCan` class instead, each can will check for itself if it collides with the ball, no matter how many cans there are in total.

To make this interaction possible, you first have to give the `PaintCan` class access to the `Ball` instance of the game. This is comparable to how you made the `Cannon` instance available. So, the trick is to give the `GameWorld` class another read-only property:

```
public Ball Ball { get { return ball; } }
```


Then, in the `PaintCan` class, you can use the expression `Painter.GameWorld.Ball` to get a reference to the ball.

8.4.1 Using Rectangular Bounding Boxes

The next step is to check whether or not a paint can is colliding with the ball. You'll need to do this inside the `Update` method of `PaintCan`—more specifically, inside the `if` block that only gets executed when the paint can is falling.

But what do we mean by “colliding”? Ideally, we'd like to check whether the sprites of the ball and the paint can are overlapping. But the paint can sprite has a pretty complicated shape, so this would be difficult to calculate. Instead, let's go for something easier: we say that there's a collision if the *rectangles* around the two sprites overlap. (In the very last part of this book, we'll look at more advanced types of **collision detection**.)

So, next question: how do you check if two rectangles overlap? The `MonoGame` engine has some nice tools that can help you answer this question. It contains a struct called `Rectangle`, which you can use to represent a rectangle with horizontal and vertical boundaries. (This is also called an *axis-aligned* rectangle.) This `Rectangle` struct has a method named `Intersects`, which can check for you if one rectangle overlaps with another. That should save you some work!

In other words, if we can describe the boundary of a `PaintCan` or `Ball` object by a `Rectangle`, we can use the `Intersects` method to do the hardest work for us. With that in mind, let's give the `PaintCan` class a new read-only property, called `BoundingBox`. This property will return the bounding rectangle around the paint can sprite:

```
public Rectangle BoundingBox { get { ... } }
```

How should you fill in this property? Well, it turns out that the `Texture2D` class already has a `Bounds` property, which gives you a `Rectangle` object representing the size of a sprite. For example, in the `Ball` class, `colorRed.Bounds` would give you a `Rectangle` that indicates the size of the red ball sprite.

But the `Bounds` property of a sprite always gives you the same rectangle: it doesn't take the *position* of the game object into account. To change that, you have to *offset* that rectangle by the object's current position. And again, you have to subtract the sprite's origin from that position, for the same reason you subtracted the origin earlier in the `PaintCan` code. Here's what the complete property should look like:

```
public Rectangle BoundingBox
{
    get
    {
        Rectangle spriteBounds = colorRed.Bounds;
        spriteBounds.Offset(position - origin);
        return spriteBounds;
    }
}
```

Go ahead and add this property to the `PaintCan` class. Add the exact same property to the `Ball` class as well: after all, we want to compare the bounding boxes of *both* types of objects.

Why the `BoundingBox` Property Works — The `Offset` method has been defined for us by `MonoGame`. It changes a `Rectangle` by adding a certain offset position to it. It's a **void** method that changes the rectangle itself, instead of returning a *new* rectangle. Luckily for us, `Rectangle`

(continued)

is a **struct**. Do you remember that struct variables are always *copies* of each other? This means that the following line:

```
Rectangle spriteBounds = colorRed.Bounds;
```

creates a copy of the sprite's bounds. You can safely edit this copy without messing up the sprite itself.

8.4.2 Responding to a Collision

With both `BoundingBox` properties in place, you're now finally ready to check if a `PaintCan` instance intersects the ball. This intersection check is given by the following expression:

```
BoundingBox.Intersects(Painter.GameWorld.Ball.BoundingBox)
```

which returns the value **true** if the two rectangles intersect and **false** if they do not intersect.

When there's an intersection, the paint can should do two things: update its color to match the color of the ball and make sure that the ball resets itself. The following piece of code does exactly that:

```
if (BoundingBox.Intersects(Painter.GameWorld.Ball.BoundingBox))
{
    Color = Painter.GameWorld.Ball.Color;
    Painter.GameWorld.Ball.Reset();
}
```

Do you see what's happening here? The `PaintCan` class is using the `Ball` class in several ways now: to get its bounding box, to get its color (and copy it into the color of `PaintCan` itself), *and* to call its `Reset` method. This is all possible, thanks to the `GameWorld` being available everywhere.

If you add this code to the `Update` method of `PaintCan` (inside the **if** block that gets executed when the can is falling), you're done! Compile and run the program again, and you'll be able to recolor the paint cans by shooting balls at them.



CHECKPOINT: Painter7

This is the final result of this chapter. Nicely done! If you're stuck somewhere, feel free to look at the `Painter7` project. We will use that project as a starting point for the next chapter.

8.5 What You Have Learned

In this chapter, you have learned:

- how objects are stored in memory and what the **null** keyword means in that context;
- the difference between values and references;
- the difference between a class and a struct;
- how to add interaction between different game objects, for example, by making the game world accessible everywhere (via a **static** property);
- how to add randomness to your game;
- how to handle basic collisions between game objects, using the `Rectangle` struct of `MonoGame`.

8.6 Exercises

1. *Static*

In this chapter, you've learned about the keyword **static**.

- What does that keyword mean? In what contexts can you use it?
- Give some examples of static properties in MonoGame that you've already seen.
- Why can't you use the keyword **this** inside a static method?

2. *Memory Models*

Assume that we have a class `MyObject` with a member variable **public int x**. Now, consider the following program fragment that uses the `MyObject` class:

```
MyObject a;  
a = new MyObject();  
a = new MyObject();  
a.x = 100;  
MyObject b = a;  
b.x = 8;  
a = null;  
b = new MyObject();
```

- For each of these instructions, draw how the objects are stored in memory *after* that instruction has been executed. (In the end, you'll have a sequence of drawings where each drawing is slightly different from the previous one.)
- Now assume that `MyObject` is a *struct* instead of a class. How does this change your answer?

3. *Random Numbers*

This question is about using the `System.Random` class.

- Create a method `RandomDouble` that takes two **double** parameters (min and max) and that returns a random **double** between those two values. (You may assume that max is larger than min.)
- Why is it common to use only *one* `System.Random` object that is shared by the entire program?
- * Can you think of examples in which you'd want to use multiple `System.Random` objects instead of one?

Chapter 9

A Limited Number of Lives



In this chapter, you're going to make the game more interesting by giving the player a limited number of lives. Whenever a paint can with the wrong color leaves the screen, the player loses a life. When the player has no more lives left, a "game over" screen will be shown, and the player gets the chance to start over.

You won't change very much in the Painter program in this chapter, but you will learn about a very important programming concept: *loops*. A loop allows you to repeat instructions a certain number of times. You'll use this to draw a number of balloon sprites in the corner of the screen, to indicate how many lives the player has left.

At the end of this chapter, you'll also learn a bit more about the scope of variables.

9.1 Maintaining the Number of Lives

Let's store the player's remaining number of lives inside the `GameWorld` class. Add another member variable to that class:

```
int lives;
```

In the `GameWorld` constructor, initialize it with a value of 5. This means that the player starts with five lives.

The `GameWorld` class is the only class that may change the `lives` variable directly. Outside that class, the only thing we want to allow is to decrease the number of lives by 1. You can add a public method for this:

```
public void LoseLife()
{
    lives--;
}
```

Remember from Chap. 4 that `lives--` is a shorter way of writing `lives = lives - 1`. So, this instruction subtracts 1 from the `lives` member variable.

When does the player lose a life? This should happen when a `PaintCan` instance leaves the screen *while having the wrong color*. Therefore, go to the `Update` method of `PaintCan`, and look for the code that

resets the paint can when it leaves the screen. Change that part of the code so that it still resets the paint can but *also* calls the `LoseLife` method if the color is wrong:

```
if (Painter.GameWorld.IsOutsideWorld(position - origin))
{
    if (Color != targetcolor)
        Painter.GameWorld.LoseLife();
    Reset();
}
```

You can compile and play the game now, but of course, you cannot yet *see* how many lives you have left. To change that, you somehow have to draw the remaining number of lives on the screen. In the Painter game, you'll do this by displaying a number of balloon sprites in the top-left corner of the screen.

Start by including the sprite *spr_lives.png* (the yellow balloon from earlier chapters) into the project and storing it inside a member variable `livesSprite`. You should be able to do this without further hints by now. We suggest to store this variable inside the `GameWorld` class.

The idea is to show five balloons if the player has five lives, four balloons if the player has four lives, and so on. With the knowledge you have so far, you could use **if** and **else** instructions for that:

```
if (lives == 5)
{
    // draw the balloon 5 times in a row
}
else if (lives == 4)
{
    // draw the balloon 4 times in a row
}
else if (lives == 3)
    // and so on
```

But this is not a very nice solution. It would require a lot of code, and you'd have to copy the same drawing instruction many times. And what if we ever wanted to start the game with *six* lives instead of five? Fortunately, there is a better solution, using a very powerful programming concept: *loops*.

9.2 Loops: Executing Instructions Multiple Times

A **loop** is a code fragment in which instructions are executed multiple times in a row. This will allow you to write only one instruction for drawing a single balloon. By adding a loop around that instruction, you can draw the sprite more often without having to write the instruction multiple times.

Looping is sometimes also called *iteration*. But the word “iteration” is also used for a single step of a loop, so that's a bit confusing. In this book, we'll use the word “loop” to refer to the overall loop and the word **iteration** for a single execution of the instruction(s) inside a loop.

There are several loop instructions in C#. In this section, we'll talk about two of them: the **while** instruction and the **for** instruction.

9.2.1 The *while* Instruction

The **while** instruction (also called a **while loop**) lets your program repeatedly execute a block of instructions. These instructions (which form the *body* of the loop) will be repeated as long as a certain *condition* holds. Just like with the **if** instruction, this “loop condition” should be a Boolean expression.

A **while** loop begins with the keyword **while**, followed by the condition between parentheses, followed by the instruction(s) to execute. As an example, take a look at the following code fragment:

```
int val = 10;
while (val >= 3)
    val = val - 3;
```

This piece of code first initializes a (local) variable. It then executes the instruction `val = val - 3`; over and over again, until the **while** condition (the expression `val >= 3`) is no longer true. What is the result of this code fragment? Well, the initial value of `val` is at least 3 (namely, 10), so the loop condition is true at first. So, the program will execute the body of the loop, subtracting 3 from the `val` variable. The loop condition `val >= 3` will then be evaluated *again*. The next value of `val` is still at least 3 (namely, 7), so the program will execute the body of the loop again, subtracting another 3 from `val`. Then, `val` will store the value 4, so `val >= 3` is still true, so 3 will be subtracted yet another time. After that, `val` will store the value 1. When the loop condition `val >= 3` gets evaluated again, it will be **false** for the first time. The program will then exit the loop and continue with the rest of the code.

Note that the body of a **while** loop can also contain more than one instruction. In that case, these instructions should be surrounded by a pair of curly braces, just like with the **if** instruction. The body of the loop can contain any instruction that you want, including method calls (or even other loops, as we'll see later).

Also, note that the condition of a **while** loop is supposed to become **false** at some point. If it stays **true** forever, then the program will also remain stuck inside that loop forever, and your game will freeze! That's still valid C#, but it's probably not what you intended to write.



Quick Reference: The while Instruction

The following code:

```
while (myCondition)
{
    ...
}
```

is a **while** loop, where ... should be replaced by your own instructions. These instructions (which are also called the *body* of the loop) will be executed over and over again, as long as the Boolean expression `myCondition` evaluates to **true**. This Boolean expression is also called the *condition* of the loop. As soon as the condition becomes **false**, the program will continue with the rest of your code.

Inside the body of the loop, something should happen so that `myCondition` will eventually become **false**. Otherwise, the code will run forever.

The example with `val` that we just gave is actually a somewhat silly way of calculating `val % 3`, the remainder after division by 3. In general, though, loops allow you to do many things that you couldn't do otherwise. For instance, in Painter, we can now use a **while** instruction to draw the number of lives on the screen:

```
int i = 0;
while (i < lives)
{
    spriteBatch.Draw(livesSprite, new Vector2(i * livesSprite.Width + 15, 20), Color.White);
}
```

```

    i++;
}

```

In this **while** instruction, the body gets executed as long as the variable *i* contains a value smaller than *lives*. Every time the body gets executed, we draw the sprite on the screen, and then we increment *i* by 1. The result of this is that we draw the sprite on the screen exactly *lives* times! So in fact, we are using the variable *i* here as a **counter**, to control how often the sprite gets drawn.

As you can see, the body of this loop contains two instructions (placed between braces). The first instruction draws the balloon sprite at a position that depends on the current value of *i*. The second instruction increases the value of *i* by one, so that the next iteration of the loop will use a larger value.

Do you see the expression *i * livesSprite.Width + 15*, which indicates the *x*-coordinate of the sprite? In the first iteration of this loop, we draw the sprite at *x*-position 15, because *i* is 0. In the second iteration, we draw the sprite at *x*-position *livesSprite.Width + 15*, because *i* is 1. In the third iteration, we draw it at *2 * livesSprite.Width + 15* and so on. If you recall that *livesSprite.Width* gives you the width of a sprite, you'll understand that this nicely draws the sprites next to each other. In total, this **while** loop draws a balloon exactly *lives* times, in such a way that you can see them all.

Note: in this example, the counter *i* doesn't only determine *how often* the instructions should be executed, but it also *changes what the instructions do*. This is a very powerful feature of loops.

We could give many more examples, but the best way to get familiar with loops is to just write them yourself. The exercises at the end of this chapter will help you with that.

For now, add this **while** loop to the Draw method of the GameWorld class. (Make sure to draw the balloons *after* the background; otherwise you won't see them.) If you compile and run the game again, you should now see five balloons in the top-left corner of the screen. As soon as you let a paint can of the wrong color leave the screen, one of these balloons will disappear. They'll keep disappearing until you have zero lives left. (What remains is to draw a nice "game over screen" as soon as that happens, but we'll get to that later in this chapter.)

9.2.2 The **for** Instruction: A Compact Version of **while**

Many **while** instructions use a counting variable and therefore have the following structure:

```

int i = beginValue;
while (i < endValue)
{
    // do something useful with i
    i++;
}

```

(Remember: *i++*; is a shorter way of writing *i += 1*;, which is (in turn) a shorter way of writing *i = i + 1*;) Because this loop structure is so common, there is actually a special kind of instruction for it: the **for** instruction.

```

int i;
for (i = beginValue; i < endValue; i++)
{
    // do something useful with i
}

```

This loop does exactly the same as the earlier **while** loop. The advantage of a **for** instruction is that everything related to the counter is nicely grouped together in the loop's header. This also reduces

the chance that you accidentally forget to write the instruction `i++`; (which would result in an endless loop).

Again, you can leave out the curly braces if the body of your loop contains only one instruction. To make the code even more compact, you can even put the declaration of the variable `i` inside the header of the **for** instruction. For example, have a look at the following code fragment:

```
for (int i = 0; i < lives; i++)
    spriteBatch.Draw(livesSprite, new Vector2(i * livesSprite.Width + 15, 20), Color.White);
```

This is the same looping instruction that draws the balloon sprite `lives` times, but it's now written in a much shorter way.

Add this instruction to the `Draw` method of `GameWorld` (instead of the **while** loop you added before), and then compile and run the game again. You should see that the game still works in the same way.

A **for** loop is a good replacement of a **while** loop whenever there's such an explicit counter involved. You usually see it in exactly that form: the header of the **for** loop clearly shows how many iterations the loop will have.

However, the header of a **for** loop can officially contain other instructions and expressions as well. In the first part of the header (before the first `;` symbol), you *initialize* the things you need before the loop starts. In the second part, you write the *condition* that needs to be true for the loop to keep running; this can be any Boolean expression, just like in a **while** loop. In the third part, you write the instruction that should always happen at the end of each iteration. Officially, this last instruction doesn't even have to be related to the counter, but it makes the most sense to write something counter-related there.

The first example of a **while** loop that we showed (the one that calculates `val % 3`) can also be written as a **for** loop, but it looks a bit weird:

```
int val = 10;
for (; val >= 3; val -= 3) {}
```

As you can see, the first part of the header is empty: there's nothing to initialize because `val` was already set to 10. The second part is the same loop condition as in the **while** version. The third part decrements the value of `val` by 3. But because this was the only thing that our **while** loop did, the **for** version doesn't need any instructions inside its body anymore!

In this example, it's pretty hard to see what the code really does. The **while** version was much easier to understand. As a rule of thumb, use a **for** loop if there's a very clearly understandable counter and a **while** loop in the other cases.



Quick Reference: The **for** Instruction

The following code:

```
for (instruction1; myCondition; instruction2)
{
    ...
}
```

(continued)

is a **for** loop, where `instruction1`, `myCondition`, `instruction2`, and ... should be replaced by your own instructions and expressions. A **for** loop can be seen as a variant of a **while** loop, and it's especially useful if your loop has a counter. The header consists of three parts:

- `instruction1` will be executed once when the loop begins;
- `myCondition` is the loop condition, a Boolean expression that gets checked in each iteration (just like in a **while** loop);
- `instruction2` will be executed at the end of each iteration.

Usually, these three things are related to a counter, like this:

```
for (int i = 0; i < someNumber; i++)
```

If your loop doesn't use a counter, it's probably better to use a **while** loop instead.

9.3 Special Cases of Loops

Let's look at a few special things that can occur when your code contains a loop.

9.3.1 Zero Iterations

Sometimes, it can happen that the condition in the header of a **for** or **while** instruction is already **false** in the very beginning. In that case, the body of the loop will not be executed at all, not even once! Look at the following code fragment:

```
int x = 1;  
int y = 0;  
while (x < y)  
    x++;
```

In this example, the condition `x < y` is immediately false, so the instruction `x++;` will never be executed. As a result, the variable `x` will keep the value 1.

9.3.2 Infinite Loops

One of the dangers of using **while** instructions (and to a lesser extent **for** instructions) is that they might never end, if you don't take care. We can easily write such an instruction:

```
while (1+1 == 2)  
    x = x+1;
```

In this case, the value of `x` will be incremented over and over again without stopping. This is because the condition `1 + 1 == 2` always yields **true**, no matter what we do in the body of the instruction. This example is quite easy to avoid, but sometimes things are a bit more subtle. Especially in **while**

instructions, you can often cause an infinite loop due to an accidental programming error. Consider the following example:

```
int x = 1;
int n = 0;
while (n < 10)
    x = x*2;
    n = n+1;
```

The intention of this code is that the value of *x* gets doubled ten times. However, unfortunately, the programmer forgot to group the last two instructions by a pair of curly braces, so only the instruction *x=x*2;* is part of the loop's body. (Remember that the compiler doesn't care about the whitespace and indentation in your program.) So, only that instruction will be repeated over and over again. The instruction *n = n+1;* will be executed when the loop finishes—but the program never gets there! That's because the condition *n < 10* never becomes **false** (because the program never changes the value of *n*).

In this case, the programmer actually should have written this:

```
int x = 1;
int n = 0;
while (n < 10)
{
    x = x*2;
    n = n+1;
}
```

So what happens if your program reaches an infinite loop? It would be a pity if you had to throw away your computer or console just because a programmer forgot to write a pair of curly braces somewhere. On a desktop computer, you can always stop the program by force (by “killing” the process that runs the program). It's hard to let the computer detect an infinite loop automatically—if a program appears to be “hanging,” it might also just be doing many complicated calculations.

As a (game) programmer, it's your job to make sure that your program doesn't contain these kinds of errors anymore once the program gets sold to customers. This is why proper testing is so important. In general, if your program seems to hang indefinitely, check out what is happening in the **while** instructions. A very common mistake is to forget incrementing the counter variable, so that the loop condition never becomes **false** and the **while** loop continues indefinitely.

9.3.3 Nested Loops

Remember that the body of a **while** or **for** loop can contain all kinds of instructions. And because a loop is also an instruction by itself, you can also write a loop *inside a loop*! We then say that these loops are **nested**. For example, take a look at this program:

```
for (int y=0; y<7; y++)
    for (int x=0; x<y; x++)
        spriteBatch.Draw(livesSprite,
            new Vector2(livesSprite.Width * x, livesSprite.Height * y),
            Color.White);
```

In this fragment, the variable *y* counts from 0 to 7 (excluding 7 itself). For each of these values of *y*, the body of the first loop gets executed. This body contains yet another **for** instruction, which we'll call the *inner* loop for now. This inner loop uses another counter *x*, which counts from 0 to the *current* value of the *y* counter. That way, the instruction inside the inner loop will be executed as many times

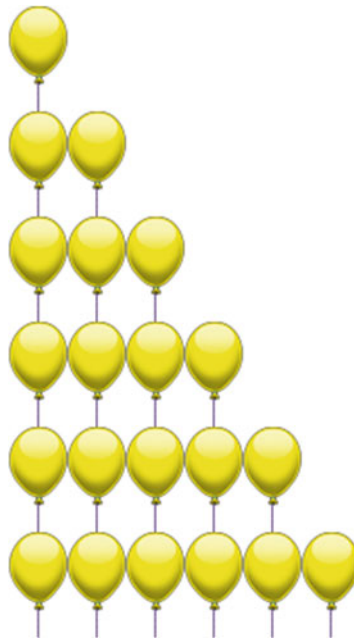


Fig. 9.1 Balloons in a triangle shape

as the *y* counter says. So, in each iteration of the outer loop, the inner loop will execute one instruction more.

As a result, this code repeatedly draws a yellow balloon sprite at the position calculated by using the values of the *x* and *y* counters. The result of this loop is a number of balloons drawn in the shape of a triangle, as in Fig. 9.1. (The first line in this shape contains zero balloons. The reason for this is that the value of *y* is still zero at that point, which means that the inner **for** instruction gets executed zero times.)

You'll see nested loops more often in the later parts of this book. They're very useful in games that are based on a *grid*, such as many puzzle games. For now, it's important to remember that the *inner* loop starts over every time, so (in this example) the counter *x* will restart at 0 in each new iteration of the *outer* loop. This has to do with the *scope* of variables inside loops. We'll talk more about that at the end of this chapter.

9.3.4 Loops Inside the Game Loop

You've seen the word "loop" quite a lot already in this book. It's important to see the difference between the loops of this chapter and the overall *game loop* that keeps the game running.

Sure, they're technically very similar: somewhere deep inside the MonoGame engine, there's (probably) a big **while** loop that makes sure that *Update* and *Draw* methods are called over and over again. But as a game programmer, you should make sure that these calls to *Update* and *Draw* don't take too much time. Otherwise, a single frame of the game will take too long to compute, and the game will not run smoothly anymore.

In “normal” software development, it’s pretty common to give a program all the time it needs to perform complex calculations. There, you can even use **while** and **for** loops to explicitly let the program wait a few seconds before going to the next instruction. In *game* development, this is very dangerous, because the game loop is already keeping track of time for you, and it wants you to respect that. For example, if you have a *Character* class and you want it to wait 5 seconds before it calls the *Jump* method, it might be tempting to write something like this:

```
float startTime = (float)gameTime.TotalGameTime.TotalSeconds;
while ((float)gameTime.TotalGameTime.TotalSeconds - startTime < 5)
{
    // wait
}
Jump();
```

But this is actually an infinite loop! Why? Because the current game time doesn’t change inside a single frame of the game loop. So the concept of “waiting” is not something you can implement in your game like this. Later in this book, you’ll learn about better ways to add a time element to your games.

If this sounds like common sense to you, then that’s a very good sign! But this topic can especially be confusing for young programmers who might already have some experience *outside* game development.

9.3.5 The Keywords *continue* and *break*

There are two more instructions in C# that allow you to do something special inside a **for** or **while** loop. First of all, the instruction **continue**; will let the program jump immediately to the *next* iteration of the loop, therefore skipping the rest of the *current* iteration. Here’s an example:

```
for (int i=0; i<10; i++)
{
    if (i % 2 == 0)
        continue;

    Console.WriteLine(i);
    ...
}
```

This program is a loop with ten iterations (with the variable *i* having a value of 0, 1, 2, ... until 9). In each iteration, we first check whether the value of *i* is *even*. If so, the program reaches the **continue**; instruction and immediately goes to the next iteration. If not, the program will execute the *Console.WriteLine* instruction (and any instructions that may come after it). As a result, this program will print the numbers 1, 3, 5, 7, and 9 to the console.

The meaning of the word “continue” may be a bit confusing here: in C#, it means “jump straight to the next loop *iteration*,” and *not* “continue to the next *instruction*.”¹

A related keyword is **break**. When the program reaches the instruction **break**; inside a loop, it will immediately jump out of this loop and continue with the next instruction *after* the loop. This keyword

¹If the latter case were true, the keyword **continue** would be rather useless!

is often used in combination with **while** loops, like this:

```
while (someCondition)
{
    ...
    if (someOtherCondition)
        break;
    ...
}
```

So, the keyword **break** offers a way to immediately stop a loop, without having to wait until the current iteration is over. As you can see, **continue** and **break** are often combined with **if** instructions, because you'll only want to skip an iteration (or leave the loop entirely) in certain special cases.

Later in this book, you'll see these keywords in action again, but you probably don't have to use them yourself very soon. For beginning programmers, it's good if you try not to use these keywords too much. Having too many **break** and **continue** keywords in your loop can make the code hard to understand, because in a way, these words "overrule" the standard behavior of a loop. Plus, you can often achieve the exact same results in other ways, such as by giving your loop a smarter loop condition.

This concludes our discussion of loops for now. Let's **continue** with the Painter game.

9.4 Adding a "Game Over" State

When the player has lost all of his/her lives in the Painter game, the game should be over. But currently, the player can still continue even when zero balloons are being drawn on the screen. In this section, you're going to show a "game over" screen instead, and you'll let the player restart the game by pressing the spacebar.

In the three main methods of the `GameWorld` class (`HandleInput`, `Update`, and `Draw`), you will let the program do different things, depending on whether or not the game is over. To make things easier, let's add a property to the `GameWorld` class that tells us whether or not the game is currently over:

```
bool IsGameOver
{
    get { return lives <= 0; }
}
```

This means that the game is not over as long as the player still has lives left. With this property in place, let's update the three methods mentioned before.

The `Update` method is the easiest one to adapt. If the game is over, none of the game objects should do anything. If the game is *not* over, the objects should behave normally. The following code achieves this:

```
if (IsGameOver)
    return;

ball.Update(gameTime);
can1.Update(gameTime);
can2.Update(gameTime);
can3.Update(gameTime);
```

Notice how this code uses a **return** instruction to exit the method earlier, preventing the rest of the code from being executed. Because `Update` is a **void** method, we can’t return a specific object, but we can still use the keyword **return** to let the method stop.

Next, let’s look at the `Draw` method. If the game is over, we want to draw a “game over” image on top of everything else. To make this possible, first include an extra sprite in the project (*spr_gameover.jpg*), and add it to the `GameWorld` class as a member variable named `gameover`. Then, in the `Draw` method, we want to draw this sprite only when the `IsGameOver` property holds. Add this code at the very end of the sprite batch, so that the “game over” image will be drawn on top of everything else. The result should look something like this:

```
public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    spriteBatch.Begin();

    // draw the ball, cannon, cans, and lives (same code as before)

    if (IsGameOver)
    {
        spriteBatch.Draw(gameover,
            new Vector2(Painter.ScreenSize.X - gameover.Width,
                Painter.ScreenSize.Y - gameover.Height) / 2,
            Color.White);
    }
    spriteBatch.End();
}
```

To center the “game over” image in the screen, we use the `Width` and `Height` properties of the `gameover` sprite, as well as the `ScreenSize` property you’ve added before. Take your time to understand why this code indeed draws the sprite in the middle of the screen.

Finally, let’s update the `HandleInput` method. If the game is *not* over, the cannon and ball should respond to the player’s input normally. If the game *is* over, the player should be able to reset the game by pressing the spacebar. The following code does this:

```
if (!IsGameOver)
{
    cannon.HandleInput(inputHelper);
    ball.HandleInput(inputHelper);
}
else if (inputHelper.KeyPressed(Keys.Space))
    Reset();
```

The `Reset` method doesn’t exist yet, so we should add it now. This method should do everything to reset the game to its initial state. So, it should set the number of lives back to five, and it should call the `Reset` methods of all game objects:

```
void Reset()
{
    lives = 5;
    cannon.Reset();
    ball.Reset();
    can1.Reset();
    can2.Reset();
    can3.Reset();
}
```

Almost there! If you now play the game a few times in a row, you might notice one last problem. The paint cans were designed to go faster and fast over time, but their *starting speed* never gets reset. So, when you restart the game by pressing the spacebar, the paint cans will still have the same base speed as when you “died” last time.

This is because the `Reset` method of `PaintCan` doesn’t reset the starting speed—and it shouldn’t, because the paint cans also call their `Reset` method during the game itself! To solve this, give the `PaintCan` class a method **public void** `ResetMinSpeed()`, which sets the `minSpeed` variable back to 30. Call that method for all three paint cans when you reset the game world. After these last changes, you’ve reached the final result of this chapter!



CHECKPOINT: Painter8

You can now fully restart the game after you’ve lost all lives.

9.5 More on the Scope of Variables

Before we finish this chapter, let’s dive further into a technical detail: the **scope** of variables. It may seem nit-picky to return to this topic again, but we’ve noticed that many new programmers are confused by the concept of scope, especially when loops are involved. Therefore, it’s a subject worth revisiting right now.

We’ve said before that the scope of a variable is limited to the smallest block of instructions (bounded by curly braces) in which that variable is declared. You’ve seen that this holds for classes and methods: it makes the difference between a member variable and a local variable. But even inside a single method, there are different levels of scope, if that method contains smaller instruction blocks again. Loops are a good example of this.

9.5.1 Variable Scope Inside a Loop

Long story short: if you declare a variable inside the body of a loop, then you can only use that variable there. Or, in more detail, the **scope** of a variable inside a loop’s body is one single iteration of that loop. At the end of an iteration, the variable will go out of scope, and it will be removed from memory. In the next iteration, a new variable with the same name will be added to the memory again.

This also means that you cannot use such a parameter anymore after the loop has finished. For example, the following code would give a compiler error:

```
for (int i = 0; i < 10; i++)
{
    int result = i;
}
result++;
```

because the variable `result` no longer exists when the program has reached a point after the loop. By the way, this is still the case if you remove the curly braces:

```
for (int i = 0; i < 10; i++)
    int result = i;
result++;
```

Even though it *looks* like the instruction `int result = i;` is no longer inside its own block, it actually still is. (But it's a block that contains only one instruction.) If you do want to use the variable `result` after the loop, you have to declare it at a higher level, like this:

```
int result;
for (int i = 0; i < 10; i++)
    result = i;
result++;
```

The counter variable `int i` is a special case: its scope is the *entire loop*. So, it doesn't get reset with each iteration (luckily!), but you can't use it anymore after the loop has finished.

9.5.2 Conclusions

It looks like we have to refine our definition of scope a bit: you already knew about member variables and local variables, but now we've discovered that "some variables are more local than others." Here's the full truth: the scope of a local variable is the smallest *block of instructions* in which that variable is declared.

A block of instructions can be the full body of a method, or the body of an `if` instruction, or the body of a loop, and so on. In fact, you can even enclose any set of instructions by curly braces, and then that set becomes its own block. For example, the following code doesn't work:

```
{
    int x = 10;
}
x++;
```

because the variable `int x` has already gone out of scope when your program reaches the instruction `x++`. But we admit that this is something you won't see very often.

Let's update the Quick Reference box from Chap. 4:



Quick Reference: Variable Scope (2)

The **scope** of a variable is the area in your code where that variable can be recognized. It determines how long the variable is kept in memory. Outside that scope, you cannot use the variable, as the compiler cannot know what value it's supposed to have.

The scope depends on where you *declare* the variable. If you declare a variable at class body level (outside any methods), it's a *member variable* that you can use in all methods of your class.

If you declare a variable inside a method, the scope of that *local variable* is the smallest *block of instructions* in which the variable is declared. This could be a complete *method*, an entire *loop* (such as a counter in a `for` loop), a single *iteration* of a loop, or any other block of instructions (usually enclosed by curly braces). Watch out for special cases like `if` and `while`: if their body is a single instruction without curly braces, then that instruction is still its own block!

9.6 What You Have Learned

In this chapter, you have learned:

- how to repeat a group of instructions in a *loop*, using the **while** or **for** instruction;
- how to use loops to draw a sprite multiple times, for example, to indicate how many lives the player has left;
- how to add a “game over” state, in which the player can restart the game;
- that the scope of a variable depends on the block of instructions in which you declare that variable.

9.7 Exercises

1. *Results of Loops*

Take a look at the following code fragments. In each code fragment, what is the value of the variable *x* after the loop has finished?

<pre>int x=0; for (int i=0; i<10; i++) x += i;</pre>	<pre>int x=0; for (int i=0; i<10; i++) if (i % 2 == 0) x += i;</pre>	<pre>int x=1; while (x < 1000) x *= 2;</pre>	<pre>int x=1; while (x > 50) x++;</pre>
---	---	---	--

2. *An Empty Loop*

Look at the following example that we used earlier in this chapter:

```
int val = 10;
for (; val >= 3; val -= 3) {}
```

This (strange) **for** loop contains an empty body with zero instructions. Why is it still required to write curly braces here? What would happen if you removed those braces?

3. *Methods with Loops*

This exercise lets you write several methods that require a **for** or **while** loop.

- Write a method `RemainderAfterDivision` that takes two **int** parameters *x* and *y*, and returns the value of *x*%*y*, *without* using the `%` operator.
- Write a method `Total` with a number *n* as a parameter that returns the total of the numbers from 0 until *n* as a result. If *n* has a value smaller than or equal to 0, the method should return 0.
- The *factorial* of a natural number is the result of the multiplication of all the numbers smaller than that number. For example, the factorial of 4 equals $1 \times 2 \times 3 \times 4 = 24$. Write a method `Factorial` which calculates the factorial of its parameter. You may assume that the parameter is always larger than or equal to 1.
- Write a method `Power` that has two parameters: a number *x* and an exponent *n*. The result should be x^n , so *x* is multiplied *n* times with itself. You may assume that *n* is a positive integer. The method should also work if *n* equals 0 and if *x* isn’t an integer number. *Hint*: use a variable for calculating the result, and don’t forget to give that variable an initial value!²

²By the way, C# already has the method `Math.Pow` that does exactly the same. You are not allowed to use that method in this exercise; that would be way too easy :)

- e. * We can approximate the so-called *hyperbolic cosine* of a real number x as follows:

$$1 + x^2/2! + x^4/4! + x^6/6! + x^8/8! + x^{10}/10! + \dots$$

In this case, the notation $6!$ means factorial of 6. Write a method `Coshyp` that calculates this approximation by summing 20 of these terms and returning that value as a result.

4. * *Prime Numbers*

For this question, it's useful to first do the exercise "Dividers" of Chap. 7.

- a. Write a method `SmallestDivider` that determines the smallest integer number ≥ 2 by which the parameter can be divided. *Hint:* try all possible dividers one by one, and stop as soon as you've found one.
- b. Write a method that determines if a number is a prime number. A prime number is a number that's only divisible by 1 and by itself.

Chapter 10

Organizing Game Objects



In Chap. 7, you've used classes to nicely group the data and behavior of the objects in your game. In this chapter, you'll improve the design of the code even more. You'll learn about one of the most powerful concepts in object-oriented programming: *inheritance*. This concept lets you use the similarities between classes, so that you don't have to write the same code in many different places.

At the end of this chapter, the code of Painter will be much cleaner, containing less duplicate code. You'll also understand how you can use inheritance in your own game projects. This will make game development easier and even more enjoyable!

10.1 Introduction to Inheritance

Before we start programming, let's take a breath and think about what it is we're going to do. What does inheritance roughly mean, and why is it useful?

10.1.1 Motivation: Similarities Between Game Objects

If you look at the different game objects in our Painter game, you will see that they have a lot of things in common. For example, the Ball, Cannon, and PaintCan classes all have very similar *member variables*: three sprites that represent each of the three different colors, a current color, a position, and often a velocity. Furthermore, they also have many *methods* in common: a Draw method that draws the appropriate sprite in the screen, often a HandleInput method for dealing with keyboard and mouse input, and often an Update method that changes the object's position and appearance. Finally, there are several useful *properties* (such as Color and BoundingBox that now appear in multiple classes.

Officially, it's not really a problem that there are so many similarities between these classes. The compiler won't complain about it, at least. However, it's a bit of a shame that you've had to copy and paste a lot of code during the process. If you ever want to add a new type of object to the game, you'll have to do even more copying and pasting.

You can also imagine that it'll be quite some work if you ever want to *change* something that is shared by many classes. For example, what if you want to add a fourth color to the game, such as

yellow?¹ Multiple classes would suddenly need a fourth sprite, and you’d have to give all classes the same extra case in their Draw method. Or what if you discover a mistake somewhere in the code? You’d have to fix the same mistake in many different places.

You can probably see where we’re going with this. As soon as multiple classes contain a lot of the same behavior, it would be nice if you could *do* something with that. That would certainly save yourself from a lot of extra work.

Here’s where the fun starts. With object-oriented programming, it’s possible to group these similarities together in a “high-level” class and then define other classes that are a *special version* of that class. This concept is called **inheritance**, named after the fact that the special cases “inherit” their behavior from the general class. Inheritance is used very often in larger programs, because it leads to understandable and maintainable code.

In Painter, you’ll create a new class that contains all the shared variables and methods that we’ve just mentioned. The classes that correspond to actual objects in your game (Ball, Cannon, and PaintCan) will inherit from this class.

10.1.2 Example and Terminology

Actually, you’ve seen one example of inheritance already. We’ve already looked at this briefly in Chap. 3: the main class of your game project (such as Painter or DiscoWorld) inherits the behavior of the Game class, which is part of the MonoGame engine. You can see this very clearly in the class header:

```
class Painter : Game
{
    ...
}
```

The header contains the name of the class itself (Painter), a colon symbol (:), and the name of another class (Game). We said before that this roughly means that Painter is a “special version” of the Game class. This allows you to reuse many game-related components (such as the game loop) so you don’t have to program those components yourself anymore.

It’s now finally time to explain the details. Let’s start by introducing the right terminology, instead of using the term “special version” all the time. There are many ways to describe the relation between Painter and Game classes:

- You can say that Painter is a **subclass** or a **child class** of Game, or that Painter *derives* or *inherits* from Game. All these words mean the same thing.
- Conversely, you can say that Game is the **superclass**, the **parent class**, or the **base class** of Painter.
- The relation between the two classes can be called a *parent-child relation*, or an *inheritance relation*.

But intuitively, you can still think of a child class as a “special version” of its parent class. An inheritance relation can best be interpreted as “is a kind of”: for example, Painter is a kind of Game, PaintCan is a kind of game object, and so on.

Also, do you notice how we’re saying “*a* subclass,” but “*the* superclass”? This is because a class can have multiple subclasses, but *a class can only have one superclass*. This is important to remember—but we’ll talk more about it near the end of this chapter.

¹Or CornflowerBlue, for old time’s sake.

10.2 Creating Your First Base Class: ThreeColorGameObject

To apply inheritance to the Painter project, let's start by creating a *base class*: a class that will contain all the functionality that is shared by Ball, Cannon, and PaintCan. We'll call this class ThreeColorGameObject, because that's what it will represent: a game object with three possible colors. The full class can be found in the Painter9 example program, but we'll work on it step by step again.

10.2.1 Member Variables

Creating the class itself is nothing special: add a new class ThreeColorGameObject to the project. Next, give it all the member variables that Ball, Cannon, and PaintCan have in common. These are (at least) the three sprites, the current color, the position, and the sprite's origin. For completeness, also add the velocity and the rotation. If an object doesn't change its position (such as the cannon) or its rotation (such as the ball), we'll deal with that later.

This is what the class should look like so far:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

class ThreeColorGameObject
{
    Texture2D colorRed, colorGreen, colorBlue;
    Color color;
    Vector2 position, origin, velocity;
    float rotation;
}
```

Later, as soon as you've specified that Ball, Cannon, and PaintCan are special kinds of ThreeColorGameObject, you can *remove* these member variables from the child classes, because they will automatically inherit the member variables of their parent. But for now, let's focus on filling the parent class further, before we start changing the child classes.

10.2.2 Methods and Properties

Next up are the methods and properties of this parent class. The *properties* are easy: Position, BoundingBox, and Color are currently exactly the same for all game objects. So, you can simply copy these properties from (say) the PaintCan class, and paste them into the ThreeColorGameObject class. (Later, we will remove these properties from the child classes again, but don't worry about that just yet.)

The *methods* are a bit less straightforward. Currently, there are (at least) four methods that some of our game objects have in common: HandleInput, Update, Draw, and Reset. But these shared methods are a bit different for each specific object. For example, PaintCan doesn't do any input handling at all, Cannon draws two sprites instead of one, and Ball has a more complicated Update method that depends on its shooting member variable.

However, even though each class has different *details* in these methods, you can still express the fact that they all *have* these methods (possibly with an empty body). The role of the ThreeColorGameObject class will be to describe the *standard behavior* of these objects. So, for each of

these four methods, `ThreeColorGameObject` will specify the instructions that all child classes have in common. Later, the child classes will be able to fill in their own details while reusing the standard behavior of their parent.

You're now going to give the `ThreeColorGameObject` class these four methods. For each of them, what should the standard behavior of a game object be? You can make several choices here, and your choice will determine what the child classes will have to do themselves later on. We suggest to fill in the parent methods as follows:

For `Reset`, the one thing that every object clearly has in common is that the color gets reset to blue:

```
public void Reset()
{
    Color = Color.Blue;
}
```

(Note: we set the `Color` property here, but we could also have set the `color` member variable instead. The advantage of using the `Color` property is that it prevents you from accidentally using an incorrect color.)

For `HandleInput`, there is no common behavior, because each object does something else (and some objects even do nothing at all). Therefore, in `ThreeColorGameObject`, this method should have an empty body:

```
public void HandleInput(InputHelper inputHelper)
{
}
```

For `Update`, the common behavior is that an object's position gets updated according to the velocity:

```
public void Update(GameTime gameTime)
{
    position += velocity * (float)gameTime.ElapsedGameTime.TotalSeconds;
}
```

Note that this also works for the objects that *don't* move, such as `Cannon`. If you simply set the velocity of the cannon to zero, then it can safely execute this instruction and nothing bad will happen.

For `Draw`, the common behavior is to first choose the correct sprite to draw (based on the current color) and to then draw that sprite with the current position, origin, and rotation. This is exactly what `PaintCan` and `Ball` currently do (except that they didn't have a changeable angle yet). Only `Cannon` does something extra, but we'll deal with that later. This is what the method should look like for `ThreeColorGameObject`:

```
public void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    // determine the sprite based on the current color
    Texture2D currentSprite;
    if (Color == Color.Red)
        currentSprite = colorRed;
    else if (Color == Color.Green)
        currentSprite = colorGreen;
    else
        currentSprite = colorBlue;

    // draw that sprite
    spriteBatch.Draw(currentSprite, position, null, Color.White,
        rotation, origin, 1.0f, SpriteEffects.None, 0);
}
```

10.2.3 Constructor

There's actually a fifth method that the child classes have (more or less) in common: the *constructor*. The constructors of Ball, Cannon, and PaintCan may not seem very "shared" because they have a different name for each class, but a lot of their work is actually the same. Let's put this overlapping work inside a constructor for the ThreeColorGameObject class. Via inheritance, we'll make sure that this "shared" constructor gets called whenever you create an instance of a child class. And just like with other methods, the child classes can still have their own constructors, in which they specify the extra work that they do themselves.

We suggest to go for a constructor with four parameters: a ContentManager object for loading sprites and the *names* of the three sprites to load. The constructor will load these sprites and store them in the Texture2D variables. The constructor will also calculate a default sprite origin and initialize the position, velocity, and rotation variables. At the end, it will call the Reset method to initialize some remaining things.

```
public ThreeColorGameObject(ContentManager content,
    string redSprite, string greenSprite, string blueSprite)
{
    // load the three sprites
    colorRed = content.Load<Texture2D>(redSprite);
    colorGreen = content.Load<Texture2D>(greenSprite);
    colorBlue = content.Load<Texture2D>(blueSprite);

    // default origin: center of a sprite
    origin = new Vector2(colorRed.Width / 2.0f, colorRed.Height / 2.0f);

    // initialize other things
    position = Vector2.Zero;
    velocity = Vector2.Zero;
    rotation = 0;

    Reset();
}
```

For all these things, the child classes will still have the option to *override* this behavior with their own specific version. For example, the Cannon class will set a different origin that does not lie at the center of the sprite. But whenever a child class does *not* explicitly want to do something else, it can freely borrow the behavior of a ThreeColorGameObject.

10.3 Turning Cannon into a Subclass

Now that you have created a parent class for game objects in general, it's time to turn the Ball, Cannon, and PaintCan into child classes of ThreeColorGameObject. In other words, you're now going to let these classes *inherit* from ThreeColorGameObject while giving them the option to add their own specific behavior as well.

You will first convert the Cannon class into a child of ThreeColorGameObject. You will also learn several new C# keywords: **virtual**, **override**, **protected**, and **base**.

10.3.1 Class Outline, Member Variables, and Properties

To officially turn the Cannon class into a subclass of ThreeColorGameObject, change the header of Cannon into the following:

```
class Cannon : ThreeColorGameObject
```

This tells the compiler that a Cannon can do everything that a ThreeColorGameObject can also do. It’s now time to clean up the Cannon class. While doing this, we should keep in mind that Cannon itself should only define the things that are *different* from a “standard” ThreeColorGameObject.

First, let’s think about what it means when we create an instance of Cannon in this new style. In the previous version of the Painter game, the Cannon instance could be depicted as in Fig. 10.1. (This is a simplified version of the figure you saw earlier in Chap. 8.)

The new version of Cannon is no longer a class that stands on its own, but it inherits from ThreeColorGameObject. When a Cannon instance gets created, this instance consists of a ThreeColorGameObject instance, plus some extras that are specific for a Cannon.

So, what are these extras? In other words, what member variables should a Cannon object add on top of a regular ThreeColorGameObject? The main extra feature of a cannon is that it draws the *barrel* as a separate sprite with a certain rotation. For this, the Cannon class needs the following additional member variables:

```
Texture2D cannonBarrel;  
Vector2 barrelOrigin;  
float barrelRotation;
```

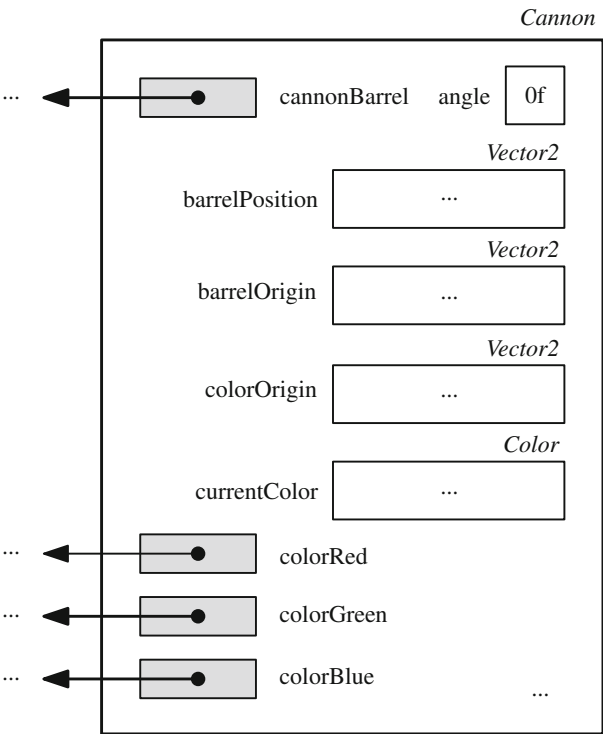


Fig. 10.1 Structure of an “old-fashioned” Cannon instance, without inheritance

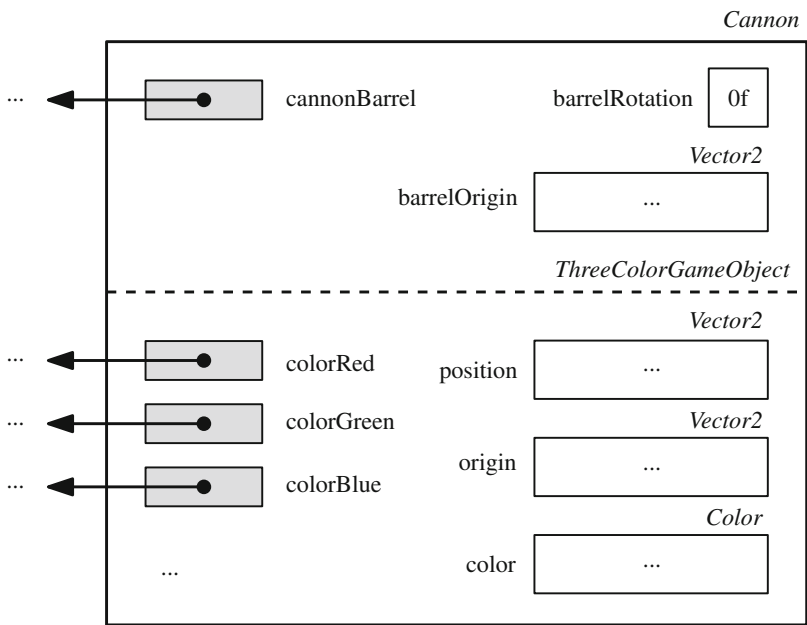


Fig. 10.2 Structure of a Cannon instance that inherits from ThreeColorGameObject

These member variables belong to Cannon and not to ThreeColorGameObject. All other member variables of Cannon can be removed now, because ThreeColorGameObject already defines them! Figure 10.2 shows how an instance of this new Cannon class would be represented in memory. We visualize it as a ThreeColorGameObject block, with the Cannon extras drawn on top of it.

To clean up the Cannon class further, remove its Position and Color properties: these are already defined by the parent class now, so you don't need to define them again here. You can also remove the Angle property because it isn't very useful anymore—it was a nice demonstration of properties in Chap. 7, but it turns out that we don't really need it. The BallPosition property should stay, because it's specific for the cannon, and the Ball class still needs to call it.

If you've followed our instructions exactly, you should see quite a few compiler errors in Cannon at this point. One reason is that some member variables have been renamed. For example, angle is now called barrelRotation, and instead of barrelPosition, we now simply use the variable position from the parent class. Fix these errors now by using the correct variable names. Have a look at the Painter9 example project if you're confused.



Quick Reference: Inheritance

To let a class *MyChild* *inherit* from another class *MyParent*, change the class header into the following:

```
class MyChild : MyParent
```

That way, each instance of *MyChild* will automatically have the same methods, member variables, and properties as *MyParent*. This allows you to reuse the code from *MyParent* without having

(continued)

to write it again for `MyChild`. Inheritance makes your code much clearer and much easier to maintain.

Watch out: inheritance only makes sense if `MyChild` is (intuitively) a *special version* of `MyParent`. Therefore, the `:` symbol can best be interpreted as “is a kind of.” Whenever a class is *not* a special kind of another class, you should not use inheritance to describe their relation.

10.3.2 Changing the Constructor

Now that we’ve defined the extra *data* of `Cannon`, let’s look at the extra *behavior*. Again, you can remove a lot of things from `Cannon` that were already defined by the parent class. To tell the compiler exactly *how* `Cannon` should reuse the behavior of `ThreeColorGameObject`, you will use several new C# keywords.

Let’s start with the *constructor* of `Cannon`. It should somehow use the constructor of its parent, because that constructor already does a lot of useful things: loading the three sprites, initializing the position and velocity, and so on. You don’t want to write this all over again for `Cannon`; that would beat the purpose of having a parent class.

In C#, there’s a special syntax for calling the parent constructor, and it’s part of the *header* of your constructor. The header of the `Cannon` constructor currently looks like this:

```
public Cannon(ContentManager Content)
```

We’d like to change this so that the constructor will also call the constructor of the *parent* class. Remember that the parent’s constructor requires four parameters: a content manager and the names of three sprites. The `ContentManager` object is something you can simply pass along, because it’s already a parameter of our current `Cannon` constructor. The three sprite names are always the same for every `Cannon` instance, namely, `spr_cannon_red` and so on. So, the list of parameters looks like this:

```
(Content, "spr_cannon_red", "spr_cannon_green", "spr_cannon_blue")
```

To call the parent constructor from inside the `Cannon` constructor, change the header of your constructor into the following:

```
public Cannon(ContentManager Content)
: base(Content, "spr_cannon_red", "spr_cannon_green", "spr_cannon_blue")
```

Here, the keyword **base** indicates that we call the constructor of the *base class*. The new line before the `:` symbol is optional; we’ve added it here to keep the code readable.

The *body* of the `Cannon` constructor should only contain the extra instructions that aren’t already handled by `ThreeColorGameObject`. So, in the `Cannon` constructor, you can now remove all instructions that the `ThreeColorGameObject` constructor already covers. The only two instructions that you need to keep are the ones that initialize `cannonBarrel` and `barrelOrigin`, because those are specifically meant for a cannon:

```
cannonBarrel = Content.Load<Texture2D>("spr_cannon_barrel");
barrelOrigin = new Vector2(cannonBarrel.Height / 2, cannonBarrel.Height / 2);
```

The `barrelRotation` variable should be set to zero, but we'll do that in the `Reset` method soon (because this is a value that can change during the game).

By the way, when this new `Cannon` constructor gets called during the game, the program will *first* execute the `ThreeColorGameObject` constructor, and *then* the extra instructions defined by `Cannon` itself. So, the work defined by the parent class will be done first, and `Cannon` defines what happens after that.

The last member variable that we're *not* yet properly setting is position. The constructor of `ThreeColorGameObject` sets this to `Vector2.Zero` by default, but a `Cannon` instance should have a different position. So, add a third instruction to the `Cannon` constructor:

```
position = new Vector2(72, 405);
```

When creating a `Cannon` instance, this variable will first be set to `Vector2.Zero` (by the parent constructor) and then to this different value (by the child constructor). This is an example of a class “overwriting” the standard behavior of its parent.

10.3.3 *protected: Making Things Available to Base Classes*

However, the last instruction that you've just added to the `Cannon` constructor is not yet allowed. This is because the `position` member variable is currently a *private* member variable of `ThreeColorGameObject`. Recall from Chap. 7 that you can use **access modifiers** (such as **public** and **private**) to indicate which parts of your code have access to the member variables, methods, and properties of a class. By default, the components of a class are *private*, unless you explicitly write the word **public** in front of them.

But we've already hinted at the existence of a third access modifier: **protected**. It's now time to finally use it. The keyword **protected** indicates that a member variable (or method, or property) can only be used by the class itself *and* by its child classes. So, to give the `Cannon` class access to the `position` member variable, you should add the keyword **protected** in front of it. In fact, you can do this for *all* member variables of `ThreeColorGameObject` that you think you'll ever want to use inside a child class. We suggest to do it for all variables except the `color`:

```
protected Texture2D colorRed, colorGreen, colorBlue;
Color color;
protected Vector2 position, origin, velocity;
protected float rotation;
```

This means that all child classes of `ThreeColorGameObject` can access and change the values stored in these variables. Note that we've kept the `color` variable private. This is because `ThreeColorGameObject` also has a nice `Color` property built around this variable, which prevents you from accidentally setting a color that isn't supported in the game. By keeping the `color` variable private, you'll force your child classes to always use this property, instead of directly using the member variable.

You can also add the keyword **protected** to methods and properties. At this point, the only *method* that you should mark as **protected** is the constructor. This is because we will never write `new ThreeColorGameObject(...)` anywhere in the code: we'll always create a `Cannon`, a `Ball`, or a `PaintCan`, but never just a `ThreeColorGameObject`. So, technically, the constructor of `ThreeColorGameObject` doesn't have to be publicly accessible. Only the *child classes* of `ThreeColorGameObject` need to have access to it. Go ahead and change the constructor's header into the following:

```
protected ThreeColorGameObject(ContentManager content,
    string redSprite, string greenSprite, string blueSprite)
```

The word **protected** makes sure that only the child classes of `ThreeColorGameObject` can call this constructor.

You'll also have to change the `Color` property of `ThreeColorGameObject` a bit. The **set** part of this property is currently **private**, to prevent any other class from changing an object's color. But now, that turns out to be a bit too strict. We'd actually like *child classes* to still be allowed to change this color, because the cannon, paint cans, and ball want to do that at several places in the code. To allow this, you should mark the **set** part of the `Color` property as **protected**, like this: The full property then looks like this:

```
public Color Color
{
    get { return color; }
    protected set
    {
        // same code as before
    }
}
```

You'll see that this fixes yet another compiler error in `Cannon`, because that class is now finally allowed to give the `Color` property a different value.

10.3.4 Overriding Methods from the Base Class

You've already updated the constructor of the `Cannon` class. Next up, let's look at the other methods of that class and see how they differ from the parent class. We'll look at the `HandleInput` method first. It currently looks like this:

```
public void HandleInput(InputHelper inputHelper)
{
    // change the color when the player presses R/G/B
    if (inputHelper.KeyPressed(Keys.R))
    {
        Color = Color.Red;
    }
    else if (inputHelper.KeyPressed(Keys.G))
    {
        Color = Color.Green;
    }
    else if (inputHelper.KeyPressed(Keys.B))
    {
        Color = Color.Blue;
    }

    // change the angle depending on the mouse position
    double opposite = inputHelper.MousePosition.Y - position.Y;
    double adjacent = inputHelper.MousePosition.X - position.X;
    barrelRotation = (float)Math.Atan2(opposite, adjacent);
}
```

The C# compiler does not automatically understand that `Cannon`'s `HandleInput` method is actually a *more specific version* of `ThreeColorGameObject`'s method. So, the compiler currently gives you a warning that `Cannon` and `ThreeColorGameObject` both have a method with the same name.

To fix this warning, you have to explicitly tell the compiler that the `HandleInput` method of `Cannon` is a special version of the parent method. In other words, you have to tell the compiler that `Cannon` *overrides* that method. To do that, add the keyword **override** to the method's header:

```
public override void HandleInput(InputHelper inputHelper)
```

But does this mean that `Cannon` can just override all parent methods without any problem? The answer is no: inside the `ThreeColorGameObject` class, you also have to be explicit about which methods are *allowed* to be overridden by child classes. For this, you need the keyword **virtual**. Only the methods that carry the keyword **virtual** in their header can be overridden. So, in the *parent* class, change the header of `HandleInput` into the following:

```
public virtual void HandleInput(InputHelper inputHelper)
```

This tells the compiler that the `Cannon` class (and any other child class) is allowed to override the behavior of this method.

While you're at it, add the **virtual** keyword to the `Update`, `Draw`, and `Reset` methods of `ThreeColorGameObject` as well. After all, these are methods that you're also going to override in the child classes. You don't have to add the word **virtual** to the constructor, because constructors work in a special way.

And finally, because you've marked all these methods as **virtual** inside the parent class, you should give them the keyword **override** in the child class. Otherwise, the compiler doesn't know that you are indeed overriding the default behavior of the parent class. Go ahead and do that to the `Update`, `Draw`, and `Reset` methods of `Cannon` now.



Quick Reference: virtual and override

The keyword **override** indicates that a method or property of a class explicitly overrides that same method or property of its base class. The keyword **virtual** indicates that a method or property can indeed be overwritten by a child class. So, you can only use **override** in the child class if you're also using **virtual** in the base class.

For example, imagine that a class `MyChild` is a child class of `MyParent`, and both classes have a method named `DoSomething`. To indicate that the child's method is a special version of the parent's method, you need to change the method headers in both classes. In `MyParent`, add the keyword **virtual**:

```
public virtual void DoSomething()
```

In `MyChild`, add the keyword **override**:

```
public override void DoSomething()
```

Without these keywords, the compiler doesn't know that these two methods are different versions of the same method.

You've seen the word **override** before, in the `Update` and `Draw` methods of the `Painter` class. We've glossed over that so far, but we can now explain that the exact same principle is used there. The standard `Game` class of `MonoGame` already defines the `Update` and `Draw` methods, and it makes sure that they are called repeatedly by the game loop. To make sure that the program will call your own

“special versions” of Update and Draw, the Painter class uses the keyword **override** here. And inside the Game class (which is part of the MonoGame engine), these methods are marked as **virtual**. You never had to worry about this yourself (because it’s already written for you in every new MonoGame project that you create), but you now understand what this code means.

Sometimes, it’s not desirable that a subclass can override any method it likes. For example, the Game class of MonoGame allows you to override its Update and Draw methods, but not the Run method. The reason is that the Run method does all kinds of low-level things that subclasses shouldn’t mess around with. By leaving out the keyword **virtual** for some methods, you can “shield” these methods from subclasses. This prevents programmers from doing serious harm to the code of the parent class.

10.3.5 Reusing Base Methods: The **base** Keyword

What is the status of the Cannon class so far? You’ve marked it as a child class of ThreeColorGameObject, so it automatically inherits the basic member variables, methods, and properties. You’ve also changed the constructor so that it reuses the constructor of the parent class. You’ve marked some elements of ThreeColorGameObject as **protected** (instead of **private**) to make sure that Cannon can access them. Finally, you’ve used the keywords **virtual** and **override** to say that the methods of Cannon are special versions of the methods of ThreeColorGameObject (and not just methods that “accidentally” have the same name).

Overriding the HandleInput method was easy, because ThreeColorGameObject itself didn’t do anything with this method yet. But the Draw and Reset methods are different: for those methods, the version in Cannon should somehow call the version of its parent, so that you can reuse the parent’s instructions. For example, the Draw method of Cannon shouldn’t contain any code that chooses a sprite depending on the current color. Instead, it should let the parent class take care of this.

To explicitly call the parent version of a method MyMethod, you can write **base.MyMethod()**. Do you remember that the keyword **this** is a special reference to the instance that is currently executing some code? Similarly, the keyword **base** also refers to the current instance, but it refers to *the part of that instance that forms the base class*.

It’s easier to explain via an example. The Reset method of Cannon should look like this:

```
public override void Reset()
{
    base.Reset();
    barrelRotation = 0;
}
```

Here, **base** refers to the current Cannon instance that has received the method call, but then “converted” to a ThreeColorGameObject with all Cannon-specific data stripped off. Think back about our drawing of a Cannon instance in memory: we drew it as a ThreeColorGameObject instance with Cannon-specific added on top of it. The **base** keyword refers to the current instance while ignoring that Cannon-specific part.

By calling **base.Reset()**, you will call ThreeColorGameObject’s version of the Reset method. This method call will reset the current Cannon instance, but it will only do the “standard” resetting that the base class knows about. Remember that this standard behavior consists of resetting the color to blue. After (or before) the method call **base.Reset()**, Cannon’s version of Reset should reset the remaining things that are specific for the cannon. In this case, the only extra thing you need to reset is the barrelRotation member variable.



Quick Reference: The Keyword **base**

The keyword **base** is comparable to the keyword **this**, but it's specifically meant for classes with inheritance.

Let's say that a class `MyChild` is a child class of `MyParent` and that it overrides the method `DoSomething`. If you want to reuse the parent's version of `DoSomething`, you can write the following:

```
public override void DoSomething()
{
    ...
    base.DoSomething();
    ...
}
```

Here, **base** refers to the `MyChild` instance that is executing the `DoSomething` method, but it is treated as an instance of `MyParent`. That way, you can call `MyParent`'s version of the method while still manipulating the current instance.

Inside every (non-static) method or property of `MyChild`, you can use '**base**.' to access all (non-private) data and behavior from the base class. Note that **base** is only defined for classes that actually *have* a base class.

You can change the `Draw` method of `Cannon` in a similar way. This method should first draw the cannon barrel (which is specific for `Cannon` objects) and then the colored sprite (which is common for all `ThreeColorGameObject` instances). In other words, it should first draw the `cannonBarrel` sprite and then call the base method:

```
public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    spriteBatch.Draw(cannonBarrel, position, null, Color.White,
        barrelRotation, barrelOrigin, 1.0f, SpriteEffects.None, 0);
    base.Draw(gameTime, spriteBatch);
}
```

This will draw the cannon in the same way as before, but you've now replaced many instructions by a single call to the base method.

The `Cannon` class is now finished! Do you see how it's much shorter than before and easier to read? Thanks to the "common code" that is already covered by `ThreeColorGameObject` now, the `Cannon` class only has to define how it is *different* from that parent class.

If you're stuck, or if you want to check your results so far, take a look at the `Painter9` example project. Focus on the `Cannon` and `ThreeColorGameObject` classes for now. We'll work on the `Ball` and `PaintCan` classes next.

10.4 Turning Ball and PaintCan into Subclasses

Using the concepts you’ve learned in this chapter, you can turn the `Ball` and `PaintCan` classes into subclasses of `ThreeColorGameObject` as well. It’s more of the same work, so we won’t go into every single detail here. The idea is (again) to remove all member variables, properties, methods, and instructions that are already defined by the parent class. Just like `Cannon`, these classes should only describe how they are *different* from their parent.

As an exercise, try to convert `Ball` and `PaintCan` into subclasses yourself. Here are a few guidelines that you can keep in mind:

- The `Ball` class only needs one member variable that its parent doesn’t have: **bool** `shooting`. This should be set to **false** in the `Reset` method. The constructor of `Ball` can then have an empty body. Note: you still need to call the base constructor, using the names of the three sprites.
- The `PaintCan` class has two extra member variables compared to its parent: `targetcolor` and `minSpeed`. These should be set in the constructor. You should also keep the instruction that sets the starting position based on the `positionOffset` parameter.
- `Ball` and `PaintCan` don’t need any *properties* of their own: the standard properties of their parent class are enough.
- The `HandleInput` method of `Ball` can stay the same, but don’t forget to add the keyword **override**.
- The `Update` method of `Ball` and `PaintCan` can stay *almost* the same, but it would be nice to let them call **base.Update(gameTime)**; instead of letting them update their position themselves. Otherwise, you’ll have three different lines of code for updating the position based on the velocity. Don’t forget to use **override** here as well!
- In the `Reset` method of both classes, use **override** again. You can reuse **base.Reset()**; now, which will already reset the object’s color to blue.
- `Ball` and `PaintCan` don’t have to override the `Draw` method: the standard behavior of their parent class is enough. You can simply *remove the entire Draw method* from these classes! This tells the compiler that `Ball` and `PaintCan` will just do exactly what their parent class says.

As usual, there is not a single “best” solution: you have quite a lot of freedom in deciding where to write what. If your game ever doesn’t do what you were expecting, compare your code to the `Painter9` example project and try to find out what you’ve done differently.



CHECKPOINT: Painter9

You’ve now improved the code of the `Painter` game by adding inheritance. Nice job!

10.5 More on Inheritance

In this section, you’ll learn more about other inheritance-related topics. You don’t immediately need these concepts in the `Painter` game, but the concepts themselves are still very important, so it’s useful to discuss them now.

10.5.1 Polymorphism

Inheritance allows you to do very interesting things. For example, because each Cannon object is now officially also a ThreeColorGameObject instance, the following instruction is allowed:

```
ThreeColorGameObject cannon = new Cannon(...);
```

This creates a new Cannon object but then stores that object inside in a *variable* of type ThreeColorGameObject! It looks a bit weird, but it *is* allowed, because one class inherits from another. Next, suppose that you would execute the following instruction:

```
cannon.Reset();
```

Which version of Reset would be called now? The version of ThreeColorGameObject or the more specific one of Cannon? Perhaps surprisingly, the answer is the Cannon version. You may not expect it, but the program “magically” knows that this ThreeColorGameObject instance is actually a Cannon. When the program is running, it will always look for the most specific version of the method to call, even if the variable has a less specific type. This effect is called **polymorphism**.

There’s one catch: to make polymorphism work, you need to use the keywords **virtual** and **override** correctly. If the Reset method of Cannon did not have the keyword **override** in its header, then the compiler wouldn’t know that this method is a more specific version of the Reset method that ThreeColorGameObject also has. In that case, the instruction `cannon.Reset();` would accidentally call the ThreeColorGameObject version of Reset!

You may wonder why this specific example is useful. Why would you ever want to store a Cannon object in a variable of type ThreeColorGameObject? Indeed, this seems a bit silly for now, but it will become very useful in the next parts of this book.

In the next games you’re going to create, the game world will no longer store one cannon, one ball, and three paint cans. Instead, it will store a *list* of game objects, and it will tell all those objects to update and draw themselves. Thanks to polymorphism, the program will automatically call the correct version of Update and Draw for each object in the list. The game world itself doesn’t have to know about the specific class type of each object. It only has to know that these objects are capable of updating and drawing themselves.

In general, the term “polymorphism” refers to the possibility to override behavior in subclasses and the program’s ability to automatically choose the most specific version of that behavior.

One example where this is useful is in the Run method that comes with every MonoGame game automatically. This method, defined in the Game class, is in charge of running the overall game loop by calling Update and Draw over and over again. The Painter class (a subclass of Game) uses the keyword **override** for its Update and Draw methods, to indicate that this class has more specific versions than Game itself. That way, thanks to polymorphism, the Run method will actually call *your* versions of Update and Draw, and not the general ones.

Another example of where polymorphism is useful is when a game company wants to release an extension of their game. For example, they might want to introduce a few new enemies or new skills that a player can learn. They can provide these extensions as subclasses of general classes such as Enemy and Skill. The actual game engine would then use these objects without having to know which particular skill or enemy it is dealing with. It simply calls the methods that were defined in the generic classes.

Why Polymorphism Is Possible — In case you’re interested, let’s look very briefly at how the compiler makes polymorphism possible. How can the program know that a variable of type `ThreeColorGameObject` actually refers to an object of type `Cannon`?

When you run the program, the compiler maintains a so-called vtable (which is short for “virtual method table”). For each object in the program, this vtable stores how that object inherits from parent classes. Whenever a **virtual** method gets called, the compiler looks in the table to find out which version of the method should be called. It will keep looking for more specific versions as long as the keyword **override** is used.

10.5.2 Hierarchies of Classes

In this chapter, you’ve added the `ThreeColorGameObject` class and you’ve let three classes inherit from it. But it doesn’t end there: a child class can have its own children, and a parent class can have its own parent again. You can add as many parent-child relationships as you want, to create an entire **class hierarchy**.

For example, imagine you want to add a special version of the ball, with extra behavior that causes it to bounce back up when it touches the bottom edge of the screen. You could write a `BouncingBall` class for this and let it inherit the behavior of the `Ball` class.

At the other end of the spectrum, you could add an even more “general” class named `GameObject`, which describes the common properties of all game objects (such as a position and a velocity). The `ThreeColorGameObject` class could then become a child of this class, inheriting the position and velocity while adding three colored sprites into the mix.

Large and complex games (or other software products) can consist of a huge hierarchy of classes. For example, consider a game that takes place in a city with a harbor. Such a game could contain many classes that somehow inherit each other’s data and behavior:

```
class GameObject { ... } // a general class for game objects with a position and a velocity
class StationaryGameObject : GameObject { ... }
class Tree : StationaryGameObject { ... }
class House : StationaryGameObject { ... }
class MovingGameObject : GameObject { ... }
class Vehicle : MovingGameObject { ... }
class MotorizedVehicle : Vehicle { ... }
class Car : MotorizedVehicle { ... }
class Truck : MotorizedVehicle { ... }
class Motorbike : MotorizedVehicle { ... }
class Bicycle : Vehicle { ... }
class Airplane : MovingGameObject { ... }
class Boat : MovingGameObject { ... }
class MotorBoat : Boat { ... }
class SteamBoat : MotorBoat { ... }
class SpeedBoat : MotorBoat { ... }
class SailBoat : Boat { ... }
```

Figure 10.3 shows the inheritance structure, using arrows to indicate an inheritance relation between classes. At the very base of the inheritance tree is a `GameObject` class. This class contains only very basic information such as the position or the velocity of the game object. For each subclass, new members (variables, methods, or properties) can be added, which are only relevant for that particular class and its subclasses. For example, a variable `numberOfWheels` should typically be in the `Vehicle` class

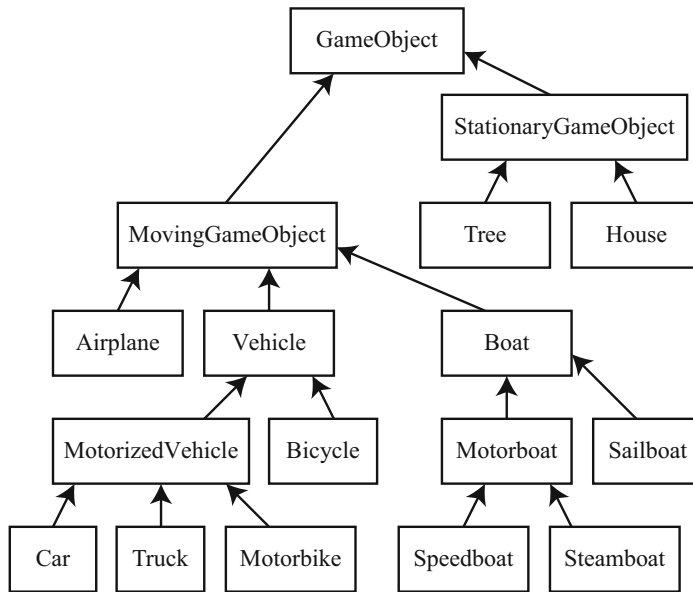


Fig. 10.3 A complicated hierarchy of classes

and not in `MovingGameObject` (because boats usually don't have wheels²). The `Airplane` class could have a variable `flightAltitude`, and the `Bicycle` class could have a variable `bellsWorking`.

Designing Class Hierarchies — When you determine how to structure your classes into a hierarchy, you have to make many decisions. In many cases, there won't be a single “best” hierarchy, but depending on the application, one design might be more useful than another.

For instance, in the example of this section, we've chosen to first divide the `MovingGameObject` class into three types of transportation: via land, air, and water. After that, we divide the classes in different subclasses: motorized or not motorized. We could also have chosen to do this the other way around: first divide the classes based on whether or not they have an engine and then on the type of transportation.

For some classes, it is not entirely clear where in the hierarchy they belong: do we say that a motorbike is a special type of bike (namely, one with an engine) or is it a special kind of motorized vehicle (namely, one with only two wheels)?

As you can see, there's not always one ultimate answer. However, the relationship between the classes themselves should always be clear. A sailboat is a boat, but a boat is not always a sailboat. A bicycle is a vehicle, but not every vehicle is a bicycle.

²Except for those really cool spy boats that can morph into a car.

10.5.3 Sealed Methods and Classes

In a large class hierarchy, it's sometimes useful to make sure that a method cannot be overridden anymore by subclasses. You can do this by adding the keyword **sealed** to the method header. For example, suppose the header of the `HandleInput` method of the `Ball` class would be given as follows:

```
public sealed override void HandleInput(InputHelper inputHelper)
```

In this case, if you ever create a subclass of `Ball` (such as `BouncingBall`), then that subclass can no longer override the `HandleInput` method. If you still try to override it, you will get a compiler error.

Likewise, you can also mark an entire *class* as **sealed**, which will ensure that programmers cannot make a subclass of that class. This can be useful if your class describes the “ultimate” version of an object and you don't want other people to create even more specific versions.

You won't need to use the keyword **sealed** yourself very soon, but if you ever come across it in someone else's code, you will at least know what it means.

10.5.4 How Not to Use Inheritance

Once again, we emphasize something very important: a class should *only* inherit from another class if the relationship between these two classes can be described as “is a kind of.” For example: a `Ball` is a kind of `ThreeColorGameObject` and `Painter` is a kind of `Game`. But by contrast, a `ThreeColorGameObject` is definitely not a kind of `Game`.

Beginning programmers often make the mistake to use inheritance to describe the relation “is a part of” instead of “is a kind of.” For example, you may think it's a good idea to create a class `Engine` that is a subclass of `MotorizedVehicle`, because all motorized vehicles have an engine. This is *not* what inheritance is meant for. Think about it: this would let the `Engine` class inherit the data and behavior of the `MotorizedVehicle` class. This would mean that an engine can suddenly have passengers, for example.

To indicate that all motorized vehicles have an engine, you should instead give the `MotorizedVehicle` class a *member variable* of type `Engine`. Member variables are a much more logical way to describe the “is a part of” relation.

Sometimes, programmers “abuse” the concept of inheritance just to give objects easy access to each other. We've seen examples where people turned `Ball` into a subclass of `GameWorld` so that it would directly inherit the screen size, for instance. This is a very, *very* bad idea. If you treat a ball as “a special type of game world” (which is really what you're doing then), then creating a new `Ball` instance would indirectly also create a new `GameWorld` instance! And because each `GameWorld` has a `Ball` as one of its member variables, each `Ball` will then contain another `Ball`. Just starting the game would no longer work properly, because you'd be creating balls inside balls, inside balls, and so on.

Long story short: use inheritance only to describe how objects are special versions of each other.

10.6 What You Have Learned

In this chapter, you have learned:

- how to turn one class into a subclass of another class, to reuse its data and behavior;
- how to override methods in a subclass to provide specific behavior for that class;

- how to use the **protected** access modifier to make data and behavior of a class available to sub-classes;
- what the difference is between **base** and **this**;
- how to use polymorphism to automatically call the right version of the method;
- how to use inheritance to create a hierarchy of related classes.

10.7 Exercises

1. *Inheritance*

Explain in your own words what *inheritance* is in a C# program. What is the main reason for using inheritance?

2. *Access Modifiers*

This question tests whether you understand the access modifiers **public**, **protected**, and **private**. Given are the following class definitions:

```
class A
{
    public float var1;
    protected int var2;
    private bool var3;

    public int Var2
    {
        get { return var2; }
        set { if (value > 0) var2 = value; }
    }

    public void MethodInA()
    {
        ...
    }
}

class B : A
{
    public int var4;
    private int var5;

    public void MethodInB()
    {
        ...
    }
}
```

a. Indicate if the following expressions are allowed inside the `MethodInA` method:

<code>this.var1</code>	<code>this.var2</code>	<code>this.var3</code>
<code>this.var4</code>	<code>this.Var2</code>	<code>base.var1</code>

- b. Indicate if the following expressions are allowed inside the `MethodInB` method:

<code>this.var1</code>	<code>this.var2</code>	<code>this.var3</code>
<code>this.var5</code>	<code>this.Var2</code>	<code>base.var1</code>
<code>base.var2</code>	<code>base.var3</code>	<code>base.Var2</code>

3. *Polymorphism*

Consider the following code:

```
class Animal
{
    public virtual void Speak() { Console.WriteLine("???"); }
}
class Dog : Animal
{
    public void Speak() { Console.WriteLine("Woof!"); }
}
class Cat : Animal
{
    public override void Speak() { Console.WriteLine("Meow!"); }
}
```

Here, `Console.WriteLine` writes the specified text to the console. Now, assume that you create instances of the classes like this:

```
Dog bingo = new Dog();
Animal bingo2 = new Dog();
Cat mittens = new Cat();
Animal mittens2 = new Cat();
```

For each of the following instructions, indicate what text gets printed on the screen.

- `bingo.Speak();`
- `bingo2.Speak();`
- `mittens.Speak();`
- `mittens2.Speak();`

The differences between these four cases are pretty tricky, so watch out! If you're confused, try reading Sect. [10.5.1](#) on polymorphism again.

4. * *Drawing the Memory*

Given are the following class definitions:

```
class One
{
    public int x;
    public One() { x = 0; }
}

class Two
{
    public One one;
    public Two(One b) { one = b; }
}

class Three : One
{
}
```

```
Two p, q;  
public Three()  
{  
    p = new Two(new One());  
    p.one.x = 7;  
    q = new Two(p.One);  
    q.one.x = 8;  
    p = new Two(this);  
    p.one.x = 9;  
}
```

Make a drawing of what the memory looks like after executing the following code:

```
Three t = new Three();
```

Just like the examples in this book, make a clear distinction between the *name* and the *value* of the variables: the name should be next to the boxes and the value inside it. Object references should be arrows, starting with a dot inside the box of the reference variable and pointing to the border of the referred object.

5. * *Type Checking*

Are the types of expressions checked during the compilation phase or when the program is running? There is an exception to this rule. In which case is this? And why is that exception necessary?

Chapter 11

Finishing the Game



In this chapter, you'll finish the Painter game by adding a few extra features, such as motion effects, sounds and music, and a score that is shown on the screen. To display the score, you'll learn about the **string** and **char** types.

The final version of the game can be found in the PainterFinal example program for this chapter.

11.1 Adding the Finishing Touches

Let's start by adding some nice extras to the game: motion effects for the paint cans, sound effects and music, and a score counter.

11.1.1 Motion Effects

In order to make the game more visually appealing, you can add a nice rotational effect to the movement of the paint cans. This will make the paint cans look like they are "swinging in the wind" a bit. To achieve this, the Update method of the PaintCan class should update the rotation member variable somehow. (And then, the Draw method will automatically draw the paint can at its newly calculated angle.)

Add the following line to the Update method (e.g., at the beginning):

```
angle = (float)Math.Sin(position.Y / 50.0) * 0.05f;
```

The Math.Sin method computes the sine of a number, which gives a nice periodic pattern for increasing input values. Because position.Y increases over time, this will give the effect of a "wiggling" sprite. The other numbers are there to make the animation look nicer: dividing the *y*-coordinate by 50.0 makes the sprite rotate more slowly, and multiplying the outcome by 0.05f makes the rotation less extreme. Feel free to play with these values to see how the behavior changes.

11.1.2 Sounds and Music

Another way to make the game more enjoyable is by adding some sound. In Chap. 5, we've explained how to add sound effects and background music to a MonoGame project. For the Painter game, we've supplied some example assets: a piece of background music (*snd_music*) and two sound effects (*snd_shoot_paint* and *snd_collect_points*). Start by including these files via the Pipeline Tool, and make sure to mark one asset as a "Song" and the other two as a "Sound Effect."

To play the background music, add the following lines to the `LoadContent` method of the main `Painter` class:

```
MediaPlayer.IsRepeating = true;
MediaPlayer.Play(Content.Load<Song>("snd_music"));
```

(You don't have to store the music file in a member variable, because you don't have to use this file anymore once the game has started. The game will just keep playing this music until you close the program.)

For the sound effects, it's a good idea to give the `Ball` class a member variable that stores the *snd_shoot_paint* sound effect and the `PaintCan` class a member variable for the *snd_collect_points* sound effect. After all, the ball knows exactly when it is being shot, and a paint can knows exactly when it has left the screen. It therefore makes sense to put these objects in charge of playing their own sounds.

Add the following member variable to the `Ball` class:

```
SoundEffect soundShoot;
```

and load it in the `Ball` constructor:

```
soundShoot = Content.Load<SoundEffect>("snd_shoot_paint");
```

Then, play the sound effect exactly when the ball is being shot. This happens inside the `HandleInput` method, exactly where you set the shooting variable to `true`. Add the following line to the `if` block that starts shooting the ball:

```
soundShoot.Play();
```

Make similar changes to the `PaintCan` class, to play a "score" sound when a paint can leaves the screen. Make sure to *only* play this sound if the can's color is correct!

11.1.3 Maintaining a Score

Scores are often a very effective way of motivating players to continue playing. (Especially *highscores* work very well in that regard, because they give players the incentive to become better than other players.) To add a score counter to the Painter game, give the `GameWorld` class the following property:

```
public int Score { get; set; }
```

Set the score to 0 in the constructor and in the `Reset` method.

Now, the idea is to add 10 points to the score each time a paint can of the correct color leaves the screen. What is the most logical place to do this? Based on what you've added so far, it makes the

most sense to do this in the `PaintCan` class, exactly at the point where you're currently playing a sound. So, the `Update` method of `PaintCan` should contain the following code fragment (among other things):

```
// reset the can if it leaves the screen
if (Painter.GameWorld.IsOutsideWorld(position - origin))
{
    // if the color is wrong, the player loses a life
    if (Color != targetcolor)
        Painter.GameWorld.LoseLife();
    // otherwise, the player earns points
    else
    {
        Painter.GameWorld.Score += 10; // this line is new!
        soundCollect.Play();
    }
    Reset();
}
```

But just like the number of lives, it would be nice if the current score is displayed on the screen. We'd like to show the score by simply drawing the text "Score: X" on the screen, where X should be filled in with the current score. This means that you somehow have to combine text ("Score: ") and a number (the current score). This is a good reason to finally take a closer look at the data types in C# for representing text: **char** and **string**.

11.2 Dealing with Text in C#

We've talked about several primitive types in C# already: for example, the numeric types such as **int** and **float** and the type **bool** for expressions that are true or false. There's another important primitive type that we've been avoiding so far: **char**. This data type is closely related to the **string** type, and they're both used for representing pieces of text.

11.2.1 The **char** Data Type

In C#, an expression of type **char** represents a single text character, such as letter, a digit, or a question mark. To indicate that something is a **char** value, you should enclose the character by *single quotes*, like this:

```
char asterisk = '*';
```

A **char** can only contain a single character. For instance, the following is not allowed:

```
char twoAsterisks = '**';
```

as that would try to squeeze two characters inside a single **char**.

Because **char** is a primitive type, variables of this type are passed by value (and not by reference), just like with **int**, **float**, and **bool**.

History of `char` — The number of different symbols that can be stored in a **`char`** variable has grown throughout the years in different programming languages:

- In the 1970s, programmers thought that $2^6 = 64$ symbols would be enough: 26 letters, 10 numbers, and 28 punctuation marks (comma, semicolon, and so on). Although this meant that there was no distinction between normal and capital letters, it wasn't a problem at the time.
- In the 1980s, people used $2^7 = 128$ different symbols: 26 capital letters, 26 normal letters, 10 numbers, 33 punctuation marks, and 33 special characters (end-of-line, tabulation, beep, and so on). The order of these symbols was known as ASCII: the American Standard Code for Information Interchange. This was nice for the English language, but it wasn't sufficient for other languages such as French, German, Dutch, Spanish, and more.
- As a result, in the 1990s new code tables were constructed with $2^8 = 256$ symbols, which included the most common letters for a particular country. The symbols from 0 to 127 were the same as in ASCII, but the symbols 128–255 were used for the special characters of a certain language. This means that there were different code tables for different languages. The West-European code table was called “Latin1.” Eastern-European languages used another table (because they have many special accents that didn't fit in the Latin1 table anymore), not to mention languages such as Greek, Russian, and Hebrew. This was a reasonable solution, but things became complicated when you wanted to use multiple languages at the same time. Also, languages containing more than 128 symbols (such as Chinese) were still impossible to represent.
- In the beginning of the twenty-first century, the coding standard was extended again to a table containing $2^{16} = 65536$ different symbols. This table could easily contain all the alphabets in the world, including many punctuation marks and other symbols. This code table is called *Unicode*. The first 256 symbols of Unicode are the same symbols as the Latin1 code table.

In C#, **`char`** values are stored using the Unicode standard.

11.2.2 The *string* Data Type

To represent a longer sequence of text (in other words, a sequence of **`char`** values), C# uses the data type **`string`**. An expression of type **`string`** should be enclosed by *double quotes*, like this:

```
string message = "Hello World!";
```

You've used strings before in the constructor of `ThreeColorGameObject`, and even in the very first `HelloWorld` example from Chap. 3. They're not very difficult to work with, but there are a few things that you need to know about.

First of all, the word **`string`** is actually an alias (a different notation) for the data type `System.String`, and `System.String` is a *class* in C#. So even though it may not look that way, **`string`** is actually a *class type*. Because **`string`** is a class type, a variable of type **`string`** can also store the value **`null`**. This happens, for example, if you declare a variable but don't initialize it yet:

```
string silence;
```

By contrast, an expression of type **char** can never be **null**, because expressions of a primitive type always have a value. By the way, a string can also have zero characters:

```
silence = "";
```

This is called the *empty string*, and it's not the same as the value **null**.

When using **string** and **char** values, you have to be careful which type of quotes you use. Single quotes are always used for a **char**, and double quotes are always used for a **string**. A string can also contain a single character, but you still have to enclose it with double quotes if you want it to have the type **string**. Also, if you don't write any quotes at all, the compiler will no longer treat your text as literal text, but as a piece of a C# program! So, there is a big difference between:

- the string "a" and the character 'a'
- the string "bool" and the type name **bool**
- the string "hello" and the variable name hello
- the string "123" and the **int** value 123
- the **char** value '+' and the operator +
- the **char** value 'x' and the variable name x
- the **char** value '7' and the **int** value 7

Predefined Types — Actually, keywords such as **int** and **float** are also aliases for more complicated terms. For example, **int** is an alias for `System.Int32`, and **float** is an alias for `System.Single`, and both of these things are actually *structs*! If you click on the word **int** or **float** in Visual Studio and press F12 ("Go To Definition"), you'll see the full definition of the associated struct. If you do the same for the word **string**, you'll see the definition of the *class* `System.String`.

Because things like numbers and text are used so often in programs, C# gives you an easier way of writing their type. It would be a bit annoying if you had to write `System.Int32` all the time. The data types that have such an alias are called the **predefined types**. But it's good to know that there's actually something more advanced behind them.

11.2.3 Special Characters and Escaping

Some text characters are a bit more difficult to represent. For example, how should you represent the single quote character if single quotes are already used *around* a character? The expression `'''` (three single quotes in a row) is invalid: the compiler can't automatically see that the middle quote is literally meant as a character, and not as the boundary of a **char** expression.

To fix this problem, you can add a *backslash* in front of the quote character. So, the expression `\"` represents the single quote character. To represent the double quote character, you need to use a similar trick, because double quotes are already used for the boundary of **string** expressions. The expression `\"` represents the single quote character.

The backslash is a special symbol that allows you to write a second symbol behind it. These two symbols together are then treated as one character. That's the only case where a **char** can be two symbols long. Putting a backslash in front of another character is also called **escaping**, because you're letting the second symbol "escape" from its default meaning.

Escaping is used for a few other special characters as well. For example, `\n` is a special character that represents a new line. When placed in a longer **string**, you can use `\n` to let the rest of your text

continue on a new line. Similarly, `'\t'` is a special character that represents a tab. There are a few more characters like this, but you won't need those very often.

Which brings us to one last problem: how should you write a backslash character if a backslash is already used for special cases? The solution is to use two backslashes in a row: the expression `'\\'` represents the backslash character.

In short, these are the five most important special characters that use escaping:

- `'\n'` for the end-of-line symbol
- `'\t'` for the tabulation symbol
- `'\\'` for the backslash character
- `'\''` for the single quote character
- `'\"'` for the double quote character

As an example, consider the following variable of type **string**:

```
string text = "I said:\n\"I'm an expert on strings!\"";
```

If you show this string on the screen (e.g., via the `Console.WriteLine` method), you will see the following:

I said:

"I'm an expert on strings!"

If your text combines a lot of quotes and backslashes, it can become quite confusing for you as a programmer to look at. Luckily, the compiler can easily tell you whether a string is well formed. Just like how the compiler will generate errors if you forget to type a `;` symbol somewhere, it will also give you errors if a string is not properly enclosed by quotes. This can happen, for example, if you forget to escape a quote character inside the text itself.

11.2.4 Operations with Strings

In many programs, you'll want to combine pieces of text or combine text with other data (such as numbers). To allow this, the `String` class has overloaded the `+` operator. You can use this operator to glue strings together into a larger string. This is also called **string concatenation**.

For example, the expression `"Hi, my name is " + "Bob"` results in the concatenated string `"Hi, my name is Bob"`. You can concatenate as many strings as you want. Of course, this becomes more interesting as soon as you start adding *variables* into the mix. For instance, let's say your game has a variable **string** `name` that stores the player's name.¹ You can then give the player a personal greeting by printing the string `"Hello, " + name + "!"`. You can also concatenate **string** and **char** values in this way.

But it doesn't end there: you can also concatenate strings and *numbers*! If you have a variable **int** `score` that stores the player's current score, then the expression `"Score: " + score` is a string that contains the current value of that variable. For instance, if the current score is 200, then the string will be `"Score : 200"`. This is exactly what you're going to do in the next section, to show the player's score on the screen in `Painter`. In the background, the number 200 will secretly be converted to a string before it is pasted into the overall result.

¹This is useful for all those people who are *not* named Bob.

Watch out: concatenation only works if at least one of the two parts is text. The following expressions will all result in the string "12":

```
"12"
"1" + "2"
1 + "2"
"1" + 2
"1" + '2'
```

But if both expressions around the + symbol are just numbers, then + will have its regular meaning of adding two numbers together. So, the following instruction:

```
string text = 1 + 2;
```

will first compute the outcome of 1 + 2 (which is the number 3) and then place that number inside a string (resulting in the string "3"). So be careful when combining text and numbers, because the + operator can have two different meanings depending on how you use it.

If you want to convert a **string** value to a **double** or an **int** value, things get a bit more complicated. This is not an easy operation for the compiler, because not all strings can easily be converted to a number. To make this easier, the primitive types have a special built-in method called `Parse`. Here is an example of how to use this special method:

```
int result = int.Parse("10");
double otherResult = double.Parse("3.14159");
```

Be careful: the `Parse` method only works if the input string actually stores a number of that type. For instance, the method call `int.Parse("haha")` will make your program crash, because the string "haha" does not contain a number of type **int**. (There are ways to prevent the program from crashing, but we won't talk about that until one of the final chapters of this book.)

There's much more that you can do with strings. In a way, a string is very similar to an *array* of characters. You'll learn all about arrays in Chap. 13, so we will get back to strings there as well.

11.2.5 Operations with Characters

For completeness, we'll also explain several interesting things that you can do with **char** expressions. You won't immediately need this in the Painter game, though.

The Unicode table (which contains all possible text characters) stores the characters in a particular order. Every symbol has its own number in this order: for example, the number of the capital letter "A" is 65, and the number for "a" is 97. Perhaps confusingly, the number for the character "0" isn't 0 but 48, because it happens to have that position in the Unicode table. The space symbol (' ') has the number 32. The character with the number 0 is a special symbol that you cannot see. (In fact, that is the default value for member variables of type **char**.)

Because characters have a nice ordering, you can use the "smaller than" and "larger than" operators to check if one character lies before or after another character in the Unicode table. For example, to check if a variable **char** `c` contains a capital letter, you could write the following expression:

```
c >= 'A' && c <= 'Z'
```

Also, because each **char** value is directly associated to a number (an index in the Unicode table), you can also place a **char** value inside an **int** variable. The **int** variable will then contain the character's

number. Consider the following code:

```
char c = 'A';
int i = c;
```

After these instructions, the variable `i` will store the value 65 (because that is the Unicode number of the character “A”). You can even do the same thing directly in one instruction:

```
int i = 'A';
```

This is always possible. After all, there are only 65536 different **char** values, while the largest **int** value is more than 2 billion.

If you want to do this conversion the other way around, you have to tell the compiler explicitly that your **int** value can safely be interpreted as a character. After all, a **char** cannot represent as many values as an **int**, so you have to let the compiler know that this is okay in your case. This means that you have to **cast** the value to a **char**:

```
int i = 65;
char c = (char) i;
```

After those instructions, the variable `c` will store the character “A.”

You can also perform calculations with symbols, if you want. Look at the following code:

```
char x = 'A';
x++;
char y = (char)(x+2);
```

The instruction `x++` replaces the value of `x` by the character that comes immediately after its current value. In this case, it assigns the character “B” because the current value of `x` is the character “A.” The expression `x+2` is an **int** expression with a value of 68, namely, the Unicode number of “B” (which is 66) plus 2. You can cast this to a **char** again to convert it to the character “D.”

11.3 Putting It to Practice: Drawing the Score

Let’s apply what you know about strings to draw the score on the screen. You will draw the score in the top-left corner of the screen, with white text on a black background image. This background image is *spr_scorebar.jpg*. Include that image in the project via the Pipeline Tool, add it as a member variable `scoreBar` to the `GameWorld` class, and draw it just after you draw the main background image:

```
spriteBatch.Draw(scoreBar, new Vector2(10, 10), Color.White);
```

To create some space for the score bar, move the yellow balloons down by changing their *y*-coordinate to 60.

To draw text inside the score bar, the `SpriteBatch` class has a method `DrawString`. However, one of the parameters of that method should be a `SpriteFont`: a special type of `MonoGame` asset that stores a *font*. We haven’t told you how to add fonts to the project yet, so let’s look at that first.

11.3.1 Adding a Font in MonoGame

For the example assets of this book, we’ve prepared a font that fits well inside the score bar. To add this font to your `MonoGame` project, open the *Content.mgcb* file in the Pipeline Tool and choose Edit

→ Add → Existing Item.... Choose the file *PainterFont.spritefont* from the *Assets/Painter* folder of this book. Then click “Build” again to include the font in the pre-built assets of your game.

If you want to create your own font from scratch instead, you could also have chosen Edit → Add → New Item.... Then choose the option “SpriteFont Description (.spritefont)” and give the file a name, such as *PainterFont*. You won’t need this option now, though.

A font file in MonoGame is actually just a text file describing some details of the font. You can’t edit this file inside the Pipeline Tool itself, but you *can* open it in any text editor (such as Notepad) by right-clicking on it and choosing Open With. If you browse through this text file, you’ll see that it contains several settings, including the font’s name, the font size, and the style such as *Regular* or *Bold*. Our pre-made file use the font Arial, a font size of 16, and the style Bold. Feel free to play around with these settings.

Now that the font is officially part of your project, add it to the *GameWorld* class as a member variable of type *SpriteFont*:

```
SpriteFont gameFont;
```

Load it in the *LoadContent* method, just like how you load sprites, but now with the *SpriteFont* data type:

```
gameFont = Content.Load<SpriteFont>("PainterFont");
```

With this font in place, you’re finally ready to use the *DrawString* method.

11.3.2 Drawing the Score on the Screen

Inside the *Draw* method of *GameWorld*, add the following line:

```
spriteBatch.DrawString(gameFont, "Score: " + Score, new Vector2(20, 18), Color.White);
```

The first parameter of this method is the font you’ve just prepared. The second parameter is the **string** you want to draw. In this case, it’s a concatenation of the string “Score: ” and the player’s current score. The third parameter is the position of the text’s top-left corner. We’ve chosen a position so that the text fits nicely inside the score bar. The fourth parameter is the color with which the text should be drawn. In this case, we’ve chosen the color white, because the score bar image is black.

Make sure to add this line *after* the line that draws the score bar; otherwise the text isn’t visible. And of course, always make sure that all draw instructions happen *before* the instruction *spriteBatch.End()*.

Compile and run the game again, and you’ll see that the text “Score: 0” is drawn in the corner of the screen. Every time you score points (by letting a paint can of the correct color leave the screen), the number after “Score: ” will change. This is because the text is redrawn in every frame of game loop, so you always see the most recent value of the *Score* property.



CHECKPOINT: PainterFinal

Congratulations: the Painter game is now finished!

11.4 Writing Documentation

In terms of code, the Painter project is finished. However, there’s one aspect we’ve been ignoring most of the time: *comments*. We’ve skipped this because it’s not the most interesting topic to talk about, but comments are still a very useful part of a program.

In general, it's good practice to use comments to explain what your methods and classes do. As we've explained in Chap. 3, you can add comments inside a method itself (to describe the meaning of your instructions). You can also add comments above the header of a method or class, to explain the overall meaning of that method or class. Programmers use the term **documentation** to refer to all comments that explain the meaning of a program.

Modern versions of Visual Studio are very good at helping you out with writing documentation. If you type three slashes (*///*) above a method or class header, then Visual Studio will create a *template* for the documentation of that method or class. For example, if you write three slashes above the header of the `GameWorld` constructor, the code will automatically look like this:

```
/// <summary>
///
/// </summary>
/// <param name="Content"></param>
public GameWorld(ContentManager Content)
{
    ...
}
```

Between the `<summary>` and `</summary>` tags, you can write a summary of what the method does. Below that, Visual Studio has added a description block for each parameter of your method. (In this case, there is one parameter, namely, `Content`.) Inside that block, you can explain what this `Content` parameter is. Here's what the full documentation could look like:

```
/// <summary>
/// Creates a new GameWorld instance.
/// This method loads all relevant MonoGame assets and initializes all game objects:
/// the cannon, the ball, and the paint cans.
/// It also initializes all other variables so that the game can start.
/// </summary>
/// <param name="Content">A ContentManager object, required for loading assets.</param>
```

If you now try to *call* this method somewhere, Visual Studio will automatically show you this text as a “hint” while you are typing. That way, you'll have the description at hand while you're programming!

In the `PainterFinal` example project, we've added documentation for all methods and classes. This indicates what a documented program should ideally look like.

It's always a good idea to make your documentation as complete as possible. This is useful if other programmers are going to look at your code. Next to that, it's also useful for *yourself*: if you look back at your program after a few weeks, you may have forgotten some of the details, especially if your project is very large. Sure, an experienced programmer can always “trace back” what a piece of code means, but this can be a very annoying process that takes a lot of time. It's better to describe the rough meaning of your code in human language as well.

Writing good documentation is a lot of work. The more complex your game project gets, the more work it will take you to keep the code fully documented. To make this a bit more bearable, it's smart to document your methods and classes as soon as possible, for example, as soon as you've finished them. That way, you don't have to write all documentation at the very end of your project. This is especially a good idea if you're working on the same project with other people, also because Visual Studio can show the documentation while a programmer is typing.

Good programmers can not only write good code; they can also explain their program clearly in natural language. This is an important thing to remember!

From now on, we won't talk about documentation explicitly anymore. As you work your way through this book, try to keep your methods and classes documented while you write them.

11.5 What You Have Learned

In this chapter, you have learned:

- how to add finishing touches to your game, such as sound effects and a score counter;
- how to use the **string** and **char** types for representing text;
- what kinds of things you can do with strings and how you can combine them with (numeric) variables;
- how to use fonts in MonoGame.
- how to add proper documentation to your code, so that programmers (including you) can understand what your program does.

This concludes Part II of the book. Up until now, you've learned about various (object-oriented) programming concepts such as **if** statements, loops, methods, classes, and inheritance. You've also learned about many MonoGame-specific concepts such as colors, sprite drawing, and input handling. You've used these concepts to create the game Painter.

If you want to extend this game further, feel free to do so! The last exercise of this chapter suggests some possible extensions, but you're free to add anything that you like.

In the next parts of this book, you will create more complicated games. These games will reuse many of the same concepts you've already seen, but they will be structured in an even better way. This general structure is something you can easily apply to other games as well. In a way, you'll be working on your own game engine inside MonoGame!

11.6 Exercises

1. *Methods with Loops and Strings*

This question lets you work towards a method that uses loops and string concatenation.

- a. Write a method `ThreeTimes` that returns three concatenated copies of a string passed as a parameter. For example, the method call `ThreeTimes("hurray!")` should result in the string `"hurray!hurray!hurray!"`.
- b. Write a method `SixtyTimes` that returns 60 concatenated copies of a string passed as a parameter. Try to limit the number of instructions in that method.
- c. Write a method `ManyTimes` that returns a number of concatenated copies of a string passed as a parameter, where that number is also passed as a parameter (you may assume that this number is 0 or larger). For example, `ManyTimes("what?", 4)` should result in `"what?what?what?what?"`.

2. *Type Conversions*

Suppose that the following declarations have been made:

```
int x;  
string s;  
double d;
```

Assume that these three variables have a value and that the variable of type **string** represents a number. Now, let's say we want to do type conversions between these variables. What should you add to the following lines of code to make this work?

```
x = d;
x = s;
s = x;
s = d;
d = x;
d = s;
```

3. * *Cuneiform*

In this advanced question, you'll create a method that uses loops and strings to convert a number to so-called cuneiform notation.

- Write a method `Stripes` with a number as a parameter (you may assume that this parameter is 0 or larger). The method should give as a result a string with as many vertical dashes as the parameter indicates. For example, the call `this.Stripes(5)` results in "|||||".
- Write a method `Cuneiform` with a number as a parameter. You may assume that this parameter will always be 1 or bigger. The method should give as a result a string containing the number in a cuneiform notation. In that notation, every number is represented by vertical dashes and the digits are separated by a horizontal dash. Horizontal dashes are also placed at the beginning and at the end of the string. Here are a few examples:

- `this.Cuneiform(25)` results in "-||-||||-"
- `this.Cuneiform(12345)` results in "-|-||-||| -||||-|||-"
- `this.Cuneiform(7)` results in "-|||||-"
- `this.Cuneiform(203)` results in "-||- -|||-"

Hint: deal with the last digit first and then repeat for the rest of the digits.

4. *Extending the Painter Game*

This challenge uses the Painter game as a basis. Here are a few suggestions of how you could make the Painter game more interesting. These are open programming exercises, so there's not one correct solution. Be as creative as you want!

- (*) Extend the game so that it costs points to shoot a ball. This challenges the player to use only as few shots as possible.
- (**) Extend the `ChooseRandomColor` method of `PaintCan` so that paint cans always receive a color that is *not* the target color. In other words, change the game so that all paint cans *must* be recolored by the player.
- (***) Change the game so that the movement of the ball can be affected by a *wind* parameter. The wind can have a direction (left or right) and a magnitude. You can implement this by giving `GameWorld` one extra member variable of type **float**. Change the ball's `Update` behavior so that it responds to the wind, but make sure that the game still stays playable! To show the current wind direction to the player, change the rotation of the paint cans as well. Also, make sure that the wind force changes a little bit over time.
- (*****) Make a two-player version of the game, where each player controls his/her own cannon. You could make a symmetrical game in which both players have to give the paint cans the right color, and the player that colors the most paint cans correctly wins the game. You could also make it an *asymmetrical* game, where the goal of one player is to correctly paint as many cans as possible, and the other player's goal is to obstruct the first player as much as possible.

Part III

Structures and Patterns

Introduction



In this part of the book, you’re going to develop your second game: *Jewel Jam*. While doing so, you will learn how to organize objects into *arrays* and *collections*. You will also learn about *recursion*: a programming technique in which a method calls itself. Furthermore, we’ll look at several game-specific topics: dealing with different screen sizes, using time in games, and creating separate classes for special tasks such as asset management. Finally, you’ll apply your knowledge of loops and *if* statements to write more complicated *game logic*.

Throughout this part, you will work on a better structure of the overall game world, so that communication between objects becomes easier. This structure will become the basis of the next games as

well. In other words, you're also going to build your own reusable "game engine" within MonoGame, so that programming other games will become much easier.

About the Game: *Jewel Jam* is a puzzle game in which the player needs to find certain combinations of jewels. The game consists of a *grid* with ten rows and five columns. Each cell of the grid contains a random jewel. The jewels are different according to three properties: they can have a different color (red, blue, or yellow) and a different shape (a diamond, a sphere, or an ellipse), and they can come in different numbers (one, two, or three jewels).

A combination of three jewels is considered *valid* if each of the properties is either the same for all jewels or different for all jewels. For example, a yellow single diamond, a blue single diamond, and a red single diamond is a valid combination: the *color* is different for all objects, and the *shape* and *number* are the same for all objects. Another example of a valid combination is a yellow sphere, a yellow double diamond, and a yellow triple ellipse: all objects have the same *color*, a different *shape*, and a different *number*. An example of a combination that is *not valid* is a yellow diamond, a red double sphere, and a blue double ellipse: the color and the shape are both different for each object, but the *number* is not the *same* for each object and also not *different* for each object. On the other hand, a yellow diamond, a red double sphere, and a blue *triple* ellipse would be a valid combination, because then the number is different for each object.

The player needs to create valid combinations of jewels in the middle column of the grid. To achieve this, the player can shift the rows of the grid around: the player can choose a different row by pressing the up and down arrow keys, and then press the left and right arrow keys to shift the jewels inside the chosen row. Whenever the player has created a valid combination in this way, he/she can press the spacebar, and then all jewels that form a valid combination will disappear. This will create empty slots in the grid's middle column. The remaining jewels will fall down to fill these empty slots, and new random jewels will fall down from the top of the screen to fill the complete column again.

If the player has created two or three combinations at the same time when pressing the spacebar, extra points will be awarded, and the game will show an overlay for some time to indicate that a double or triple combination has been made.

During the game, a jewel cart is slowly moving away; once this cart has left the screen, the player's time is up. Every time the player makes valid combinations of jewels, he/she gains points and the jewel cart moves back a few steps (thus giving the player some extra time). The overall goal is to score as many points as possible before the time runs out.

If you already want to play the complete version of the game, to get a feeling of how the game is supposed to work, open the solution belonging to Chap. 16 and run the *JewelJamFinal* project.

Chapter 12

Dealing with Different Screen Sizes



In this chapter, you will learn how to scale your game to different screen sizes. You will learn how to change the size of the game’s window, how to scale the game world so that it fits inside this window, and how to make your game run in full-screen mode. We choose to do this at the beginning of this part, because the Jewel Jam game contains large images that may not fit on all screens.

Dealing with screen sizes is a somewhat “dirty job” that simply needs to happen before we can continue. If you want, you can go through this chapter more quickly and take the JewelJam1c example project as a starting point for the next chapter. We still strongly encourage you to write the program yourself, though.

Also, Sect. 12.4 is very useful in particular, because it discusses something you’ll see very often in game programming: the difference between *screen* coordinates and *world* coordinates.

At the end of this chapter, you will have a program that shows the background image of Jewel Jam scaled to the chosen window size. You will also have a nice reusable piece of code for the games you’ll create later on.

12.1 Scaling the Game World to Fit in the Window

Start by creating a new MonoGame project called *JewelJam*. You’ll be extending this project throughout this part of the book. Rename the `Game1` class to `JewelJam`, and feel free to remove its `Initialize` and `UnloadContent` methods. Also, remove the namespace around each of the classes that Visual Studio creates automatically. Just like in the previous part of the book, we won’t be using namespaces just yet.

All assets for Jewel Jam can be found in the *Assets/JewelJam* folder of this book’s supplementary files. Start by including one sprite via the Pipeline Tool: *spr_background.jpg*. In the `JewelJam` class, load this sprite in the `LoadContent` method, store it in a member variable called `background`, and draw it in the `Draw` method. If you’ve forgotten how to do these things, have a look at example projects from the previous chapters.

If you compile and run the project now, your game should look like Fig. 12.1. As you can see, only a small part of the background is visible. This is because the background sprite is much larger than the standard window size by MonoGame.

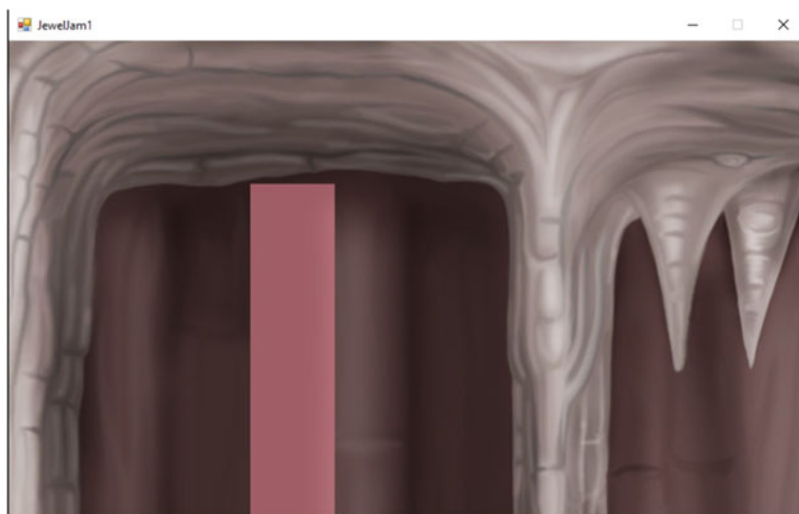


Fig. 12.1 A screenshot of your first version of Jewel Jam

12.1.1 Changing the Window Size

To change the window size so that it matches the size of the background sprite, you can add the following lines to the `LoadContent` method. (Make sure to add these lines *after* you’ve loaded the background sprite; otherwise, expressions such as `background.Width` cannot be evaluated.)

```
graphics.PreferredBackBufferWidth = background.Width;
graphics.PreferredBackBufferHeight = background.Height;
graphics.ApplyChanges();
```

`PreferredBackBufferWidth` and `PreferredBackBufferHeight` are properties of the `GraphicsDevice` class. These properties allow you to get or set the width and height of the window. After you’ve changed these values, you need to call the `ApplyChanges` method to let your changes take effect.

Compile and run the game again. You should now see a much larger window in which the background sprite fits perfectly. But if your computer screen doesn’t have a very high resolution, you won’t be able to see the full window. And because you can’t predict what screen resolutions your *players* will have, it would be nice to do something more advanced.

12.1.2 Storing the World Size and the Window Size Separately

To improve this, you’re going to choose a smaller window size, and you’re going to *scale* the background sprite so that it fits inside this smaller screen. This means that there are two different sizes in your game: the size of the *window* and the size of the *game world*. Let’s create two member variables for this, both of type `Point`:

```
Point worldSize, windowSize;
```

`Point` is a struct that belongs to the MonoGame library. It’s similar to `Vector2` in many ways, but the main difference is that it stores two *integers* instead of two floats. This makes it useful for representing things like the size of the window, or the size of a sprite, which can never contain “half a pixel”

anyway. In the LoadContent method, *before* you try to resize the window itself, initialize these variables as follows:

```
worldSize = new Point(background.Width, background.Height);  
windowSize = new Point(1024, 768);
```

This means that we're going to fit the game world inside a window of 1024×768 pixels and that the game world itself is exactly as large as the background sprite.

To resize the window and calculate how the world should be scaled, create a separate method with the following header:

```
void ApplyResolutionSettings()
```

In this method, start by changing the window size to match the `windowSize` variable:

```
graphics.PreferredBackBufferWidth = windowSize.X;  
graphics.PreferredBackBufferHeight = windowSize.Y;  
graphics.ApplyChanges();
```

12.1.3 Calculating How to Scale the Game World

Next, we should calculate how the game world needs to be stretched or shrunk to fit exactly inside that window. How should we do that? Well, for example, if the game world is twice as *wide* as the window, it should be scaled *horizontally* by a factor of 0.5. Or if the game world is only half as *high* as the window, it should be scaled *vertically* by a factor of 2. In general, the following expressions will give you the scaling factors you're looking for:

```
windowSize.X / worldSize.X  
windowSize.Y / worldSize.Y
```

Take your time to realize why this is correct. (You don't have to add these expressions to the program yet; we'll do that later.)

There's one catch: `windowSize.X` and `worldSize.X` are both integers, and the result of a division between integers in C# is always an integer again. So, the result of this division will be rounded down to the nearest whole number. This means that we can never receive a scaling factor of 0.5, for example. To avoid this, you need to *cast* at least one of the two numbers to a **float**. Dividing a **float** by an **int** (or the other way around) will give a result of type **float**. For example, you can cast the `windowSize` part of the expressions, as follows:

```
(float>windowSize.X / worldSize.X  
(float>windowSize.Y / worldSize.Y
```

Now, the idea is that the Draw method should use these two scaling factors to draw all sprites a bit smaller or larger. To make this possible, there's a different version of `spriteBatch.Begin` that allows you to add a **scaling matrix**. If you specify this matrix before you start drawing any sprites, then all sprites will be scaled by that matrix.

We won't talk too much about how a matrix works; it's a fairly difficult mathematical topic that doesn't really fit in this book. Instead, we'll just show you how to use this concept in your game. First, add a new member variable of type `Matrix`:

```
Matrix spriteScale;
```

A `Matrix` is another `MonoGame` struct, just like `Vector2` and `Color`. It's exactly the type of object that the different version of `spriteBatch.Begin` needs.

At the end of your `ApplyResolutionSettings` method, initialize this matrix as follows:

```
spriteScale = Matrix.CreateScale(
    (float>windowSize.X / worldSize.X, (float>windowSize.Y / worldSize.Y, 1);
```

As you can see from the syntax, `CreateScale` is a *static* method of `Matrix`, because you type the *class name* in front of the dot (and not a specific instance name). This method gives you a new `Matrix` object that you can store in the `spriteScale` variable. We won't bother you with the details of this method—simply put, it gives you the correct scaling matrix based on the scaling factors that you give to it as parameters. Do you notice that it contains the two scale factors we've talked about earlier?

The `ApplyResolutionSettings` method is now finished. Make sure to call this method at the end of `LoadContent`:

```
ApplyResolutionSettings();
```

Also, remove any old code of `LoadContent` that tried to scale the screen. After all, scaling-related tasks are now completely handled by your new method!

12.1.4 Scaling the Game World

The final step is to use the new `spriteScale` member variable for actually scaling the sprites. In the `Draw` method, replace the following line:

```
spriteBatch.Begin();
```

by this line:

```
spriteBatch.Begin(SpriteSortMode.Deferred, null, null, null, null, null, spriteScale);
```

This calls the different version of the `spriteBatch.Begin` method, where `spriteScale` is one of its parameters. Don't worry about the other parameters for now. These are other settings that we don't need yet at this point in the book.

Compile and run the game to see the new results: the background image is squeezed into a smaller screen. In fact, if you would draw other sprites on top of the background, they would receive the same scaling effect: the *entire game world* is shown in a stretched way.



CHECKPOINT: JewelJam1a

12.2 Making the Game Full-Screen

The game world is now scaled to fit inside the window you've specified. But it would be even nicer if we could also let the game run in *full-screen mode*: that way, players with larger screens can see the game in its full glory, without their desktop or other programs being visible in the background. In this section, you'll add this full-screen mode, and you'll allow players to turn that mode on and off.

It's actually really easy to make the game run in full-screen mode. Simply add the following instruction just before you call `graphics.ApplyChanges()`:

```
graphics.IsFullScreen = true;
```

But it's better to *not* run the game now, because it has become difficult to quit the game: there's no icon for closing the window anymore. To improve that, let's add an option for *toggling* between full-screen mode and windowed mode.

12.2.1 Allowing Full-Screen Mode and Windowed Mode

First, let's change the `ApplyResolutionSettings` method so that it allows both full-screen mode and windowed mode. Give this method a parameter: **bool** `fullScreen`. Depending on the value of this parameter, the method will set the graphics settings a bit differently. At the beginning of this method, set the `IsFullScreen` property to match this new parameter:

```
graphics.IsFullScreen = fullScreen;
```

Next, you should determine which screen size to use. In windowed mode (when `fullScreen` is **false**), the screen size is simply the desired window size. In full-screen mode (when `fullScreen` is **true**), the screen size is the full-screen resolution. The following two expressions will give you the width and height of the full screen:

```
GraphicsAdapter.DefaultAdapter.CurrentDisplayMode.Width  
GraphicsAdapter.DefaultAdapter.CurrentDisplayMode.Height
```

To correctly determine which screen size to use (based on the `fullScreen` parameter), add the following code to the beginning of your `ApplyResolutionSettings` method:

```
Point screenSize;  
if (fullScreen)  
{  
    screenSize = new Point(  
        GraphicsAdapter.DefaultAdapter.CurrentDisplayMode.Width,  
        GraphicsAdapter.DefaultAdapter.CurrentDisplayMode.Height);  
}  
else  
    screenSize = windowSize;
```

In the rest of your method, you should now use `screenSize` instead of `windowSize`. This is because scaling based on `windowSize` no longer makes sense when you're running in full-screen mode. If you use `screenSize` instead, the program will apply the correct scaling in full-screen mode *and* in windowed mode. So, the rest of the method should look like this:

```
// scale the window to the desired size  
graphics.PreferredBackBufferWidth = screenSize.X;  
graphics.PreferredBackBufferHeight = screenSize.Y;  
  
graphics.ApplyChanges();  
  
// calculate how the graphics should be scaled  
spriteScale = Matrix.CreateScale(  
    (float)windowSize.X / worldSize.X, (float)windowSize.Y / worldSize.Y, 1);
```

In the `LoadContent` method, you should now pass the value **true** or **false** as a parameter for the method `ApplyResolutionSettings`. Start by giving it the value **false**, and then compile and run the game again. You will see the same thing as before: a game in windowed mode, where the background is scaled to fit inside the window.

If you give the method the value **true** instead, the game will run in full-screen mode—but be aware that the game will (again) be a bit difficult to close then.

12.2.2 Toggling Between the Two Modes

Now, let's give players the option to switch between windowed mode and full-screen mode. We'll switch between modes when the player presses the F5 key. To make this easier, copy the `InputHelper` class from your `Painter` project into this project as well. This class has nice methods for checking whether a keyboard key has been pressed.

Just like in the `Painter` project, give the `JewelJam` class a new member variable:

```
InputHelper inputHelper;
```

Initialize it in the constructor of `JewelJam`:

```
inputHelper = new InputHelper();
```

And make sure that the input helper is updated in each frame of the game loop:

```
protected override void Update(GameTime gameTime)
{
    inputHelper.Update();
}
```

Now for the actual toggling, start by giving the `JewelJam` class the following property:

```
bool FullScreen
{
    get { return graphics.IsFullScreen; }
    set { ApplyResolutionSettings(value); }
}
```

Can you see what's happening here? The **get** part of this property simply returns the current value of the expression `graphics.IsFullScreen`, meaning whether or not the game is currently in full-screen mode. The **set** part calls your own `ApplyResolutionSettings` method, where **value** indicates whether or not the game should run in full-screen mode. So, if the program ever reaches the instruction `FullScreen = true`; or `FullScreen = false`;, the game will go to either full-screen mode or windowed mode.

What remains is to *use* this new property in the program, to toggle between modes. At the end of `LoadContent`, replace the following instruction:

```
ApplyResolutionSettings(false);
```

by this:

```
FullScreen = false;
```

This won't change the program's behavior, but it looks nicer because the work is now hidden behind the `FullScreen` property.

And in the `Update` method, change the value of the `FullScreen` property when the player has just pressed the F5 key:

```
if (inputHelper.KeyPressed(Keys.F5))
    FullScreen = !FullScreen;
```

(You'll need to include the `Microsoft.Xna.Framework.Input` library for this.) Note how the second instruction "flips" the value of the `FullScreen` property: it gives that property the exact opposite value of the one it currently has.

Compile and run the game again. The game will start in windowed mode. Whenever you press F5, you will switch to full-screen mode or vice versa. Nice work!

12.2.3 Allowing the Player to Quit the Game

As a final touch, you can use the `InputHelper` to close the game when the player presses the Escape key. To do that, add the following lines to the `Update` method:

```
if (inputHelper.KeyPressed(Keys.Escape))  
    Exit();
```

The `Exit` method is part of the `Game` class (so the `JewelJam` class automatically *inherits* this method). Calling this method will cause the game to close itself. Similar instructions were actually already part of the initial project that Visual Studio created for you. At this point, feel free to remove those instructions and replace them with our own version.



CHECKPOINT: JewelJam1b

Compile and run the game, and verify that you can now indeed toggle full-screen mode and quit the game.

12.3 Maintaining the Aspect Ratio

One problem of our solution so far is that it doesn't check if the game world is stretched equally in the horizontal and vertical directions. For example, if you choose a window size of 1000×200 pixels instead, the background will be stretched in a very weird way in windowed mode. Instead, it would be nicer to preserve the **aspect ratio** of the background image (the ratio between its width and height). In simpler words, we don't want to stretch the game world out of proportion.

In this section, you will extend your `ApplyResolutionSettings` method to do exactly that. Unfortunately, the solution involves quite a lot of code, and it's a bit tricky to understand. We can't blame you if you get lost somewhere along the way. If that happens, you can also look at the `JewelJam1c` example project for the final result.

12.3.1 What Is a Viewport?

In official terms, you're going to calculate the **viewport** in which the game will be shown. A viewport is a certain rectangle *within* the game window in which you want to draw the game. The viewport for your game is possibly smaller than the entire window, but never larger.

What should this viewport look like? It should be the largest possible rectangle that fits in the window *without* stretching the game world out of proportion. The viewport should also be nicely *centered* inside the game window. Figure 12.2 shows the possible cases that can occur:

1. (Left image) If the window is relatively wide, then the game world can be stretched out to use the window's full height. But it cannot use the window's full *width*, because then we would stretch the game world more horizontally than vertically. Instead, we want to show two black bars on the left and right side.
2. (Right image) If the window is relatively high, then the opposite occurs: the game world can use the window's full width, and we want to show two black bars at the top and the bottom.
3. If the game world and the window have exactly the same aspect ratio, then the game world can simply fill the entire window, and it won't get stretched out of proportion.

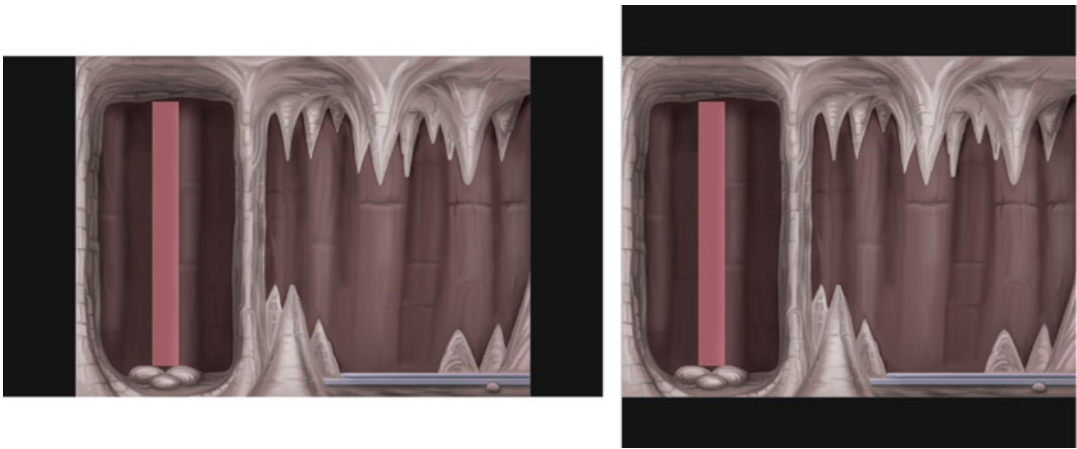


Fig. 12.2 Two ways in which the game could fit into a window without stretching out of proportion

12.3.2 Calculating the Viewport

To calculate the viewport, let's add a new method `CalculateViewport` to the `JewelJam` class. This method has the following header:

```
Viewport CalculateViewport(Point windowSize)
```

`Viewport` is a special `MonoGame` class for describing a viewport. Your method should *return* an instance of that class. The method takes the window's full size as a *parameter*. Also, the method doesn't have to be **public** because only the `JewelJam` class itself will use it.

Inside this method, you can start by creating a new `Viewport` object:

```
Viewport viewport = new Viewport();
```

Before you return this viewport, you need to set four of its properties: `X`, `Y`, `Width`, and `Height`. The properties `Width` and `Height` represent the *size* of the viewport. The properties `X` and `Y` represent the *top-left corner* of the viewport inside the game window. We need to calculate these four values now.

We will first calculate what the *size* of the viewport should be. Remember: the final goal is to calculate the largest possible viewport that fits inside the window. To get there, first compute the aspect ratio of the *game world* and the aspect ratio of the *window*, as follows:

```
float gameAspectRatio = (float)worldSize.X / worldSize.Y;
float windowAspectRatio = (float>windowSize.X / windowSize.Y;
```

You can use this to find out if we are in case 1, 2, or 3 from the list we've just described. We'll just go ahead and give you the solution:

```
// if the window is relatively wide, use the full window height
if (windowAspectRatio > gameAspectRatio)
{
    viewport.Width = (int)(windowSize.Y * gameAspectRatio);
    viewport.Height = windowSize.Y;
}
// if the window is relatively high, use the full window width
```

```

else
{
    viewport.Width = windowSize.X;
    viewport.Height = (int)(windowSize.X / gameAspectRatio);
}

```

This code sets the width and height of the viewport to the correct values. In the first **if** case, the window is too wide (case 1 of the list), so our viewport can use the window's full height (`windowSize.Y`). For the *width*, we compute a matching value so that the game world maintains its aspect ratio. In the **else** case, the exact opposite occurs (case 2 of the list).

This code does *not* include case 3 of the list, in which the game world and the window have exactly the same aspect ratio. That's because the **else** case automatically covers it as well. Feel free to take a moment to understand why this is true.

To set the *top-left corner* of the viewport, add the following two instructions:

```

viewport.X = (windowSize.X - viewport.Width) / 2;
viewport.Y = (windowSize.Y - viewport.Height) / 2;

```

As an exercise for yourself, try to understand why this code is correct. To give you some intuition: if the viewport fills the window's full width, then `windowSize.X` and `viewport.Width` are equal, so the first instruction will place the viewport at *x*-coordinate 0 (which is the left side of the window). If the window is much wider than the viewport, the *x*-coordinate of the viewport will become larger, so the viewport will lie more to the right. The “/ 2” part makes sure that the extra space is nicely divided into two parts.

To conclude the method, return the viewport object that you've now filled in.

12.3.3 Using the Viewport for Drawing Sprites

Now go back to your `ApplyResolutionSettings` method. Replace this last instruction:

```

spriteScale = Matrix.CreateScale(
    (float)screenSize.X / worldSize.X, (float)screenSize.Y / worldSize.Y, 1);

```

with these two instructions:

```

GraphicsDevice.Viewport = CalculateViewport(screenSize);
spriteScale = Matrix.CreateScale(
    (float)GraphicsDevice.Viewport.Width / worldSize.X,
    (float)GraphicsDevice.Viewport.Height / worldSize.Y,
    1);

```

What does this do? The first instruction calls your new `CalculateViewport` method and then applies the result of that method to the graphics device. This will have the effect that the `Draw` method will draw all sprites inside your viewport and leave the rest of the screen blank. But because of this, you also have to change how the sprites are *scaled*. That's why you have to change the second instruction as well: instead of scaling the sprites to fit on the *screen*, you now have to scale them to fit in the *viewport*.

Compile and run the program again, and press F5 to switch to full-screen mode. If you have a wide computer screen, you should now see two colored bars at the sides. The game world is no longer

getting stretched out of proportion. If you want to test this some more, you could also try out a very high window, such as this:

```
windowSize = new Point(500, 800);
```

In windowed mode, you'll then see bars at the top and bottom of the window. The game world will still have the correct aspect ratio.

Note: the color of the side bars is the color that you give to `GraphicsDevice.Clear` as a parameter. That color is probably still `Color.CornFlowerBlue` at this point, because that's the default value in every new MonoGame project that you create. Feel free to change this color now. We suggest to use `Color.Black` because it gives the least distracting image.¹

12.4 Screen Coordinates and World Coordinates

You now have a game that nicely scales to match the screen size, without stretching out of proportion. But this scaling has led to something new and interesting. Now that we allow our game to stretch out, the position of an object on the *screen* is no longer the same as its position in the *game world*.

In other words, we've now created a difference between *screen coordinates* and *world coordinates*. The **screen coordinates** of an object indicate the object's pixel position, measured from the top-left corner of the window. The **world coordinates** of an object indicate the object's actual position in the game world. These two things are not the same! For example, imagine a game world of 50×50 pixels stretched to a window of 500×500 pixels. In that case, a *screen* position of (100, 100) would correspond to a *world* position of (10, 10).

In the Painter game, there wasn't any difference between screen coordinates and world coordinates, because the game world was exactly as large as the window. In Jewel Jam (and the later games of this book), there *will* be a difference. If the size of the window changes (such as when you switch to full-screen mode), the screen coordinates of objects will change, but their world coordinates will stay the same. The window will then simply show a different "picture" of the game world, without changing the game world itself.

Why are we telling you this? Well, sometimes you need to be aware of this difference. An important example is when you add *mouse interaction*. When you ask MonoGame for the position of the mouse pointer, you'll receive that position in *screen coordinates*: a pixel position relative to the top-left corner of the window. If the player clicks the left mouse button, and you want to find out where the player has clicked in the *game world*, you'll have to translate the mouse position to *world coordinates*. This is called **coordinate conversion**.

Imagine what would happen if you did *not* translate the mouse position to world coordinates. Take the Painter game as an example, where players can shoot a ball by clicking on the screen. If you would stretch the game to a big 4K screen, then the *pixel* distance between the mouse and the cannon can become much larger. As a result, players will be able to shoot the ball much faster! This is not something we want. It's OK if the game looks nicer on a bigger screen, but the *gameplay* shouldn't change.

¹Nobody's stopping you from creating DiscoWorld side bars, though.

12.4.1 Converting Screen Coordinates to World Coordinates

To account for the difference between screen and world coordinates, let's give the `JewelJam` class a method that transforms screen coordinates to world coordinates. The method looks like this:

```
Vector2 ScreenToWorld(Vector2 screenPosition)
{
    Vector2 viewportTopLeft =
        new Vector2(GraphicsDevice.Viewport.X, GraphicsDevice.Viewport.Y);

    float screenToWorldScale = worldSize.X / (float)GraphicsDevice.Viewport.Width;

    return (screenPosition - viewportTopLeft) * screenToWorldScale;
}
```

This method takes a position in screen coordinates as a parameter and returns the matching position in world coordinates. In summary, the method scales the screen position with the same scaling that the `Draw` method also uses. It also takes into account that the top-left corner of the *viewport* isn't always the same as the top-left corner of the *screen*.

12.4.2 Showing the Mouse Position

To verify that your method works correctly, let's draw a sprite at the mouse position. First, in the constructor of `JewelJam`, make sure that the “regular” mouse cursor is visible:

```
IsMouseVisible = true;
```

This will allow us to easily check our program. Next, choose a small image from the `JewelJam` example files (such as one of the jewel sprites), include it in your project via the Pipeline Tool, and store it in a member variable named `cursorSprite`.

Then, in the `Draw` method, draw your sprite at the current mouse position. (Make sure to draw the sprite *after* you draw the background.) Let's see what happens if you *don't* use the `ScreenToWorld` method:

```
spriteBatch.Draw(cursorSprite, inputHelper.MousePosition, Color.White);
```

Choose a strange window size, such as 500×800 pixels, and then compile and run the game again. If you move the mouse pointer around, you'll see that the positions of the cursor and your sprite don't match. This is because `inputHelper.MousePosition` gives you the mouse position in *screen* coordinates, and you're using that position directly in the game world. Because the game world is being scaled to fit in a smaller screen, the sprite's position does not correctly represent the mouse position anymore.

The solution is to first translate the mouse position to *world* coordinates:

```
spriteBatch.Draw(cursorSprite, ScreenToWorld(inputHelper.MousePosition), Color.White);
```

This will draw the cursor sprite at the *world position* that the mouse is currently pointing at. Compile and run the game, and you'll see that the sprite is now nicely aligned with the real mouse pointer.



CHECKPOINT: JewelJam1c

What About WorldToScreen? — In a way, the Draw method implicitly converts world coordinates to screen coordinates. The Matrix object that you’ve created more or less acts a WorldToScreen method, which does exactly the opposite of your own ScreenToWorld method: it moves and scales world objects in such a way that they fit on the screen.

By adding ScreenToWorld in your last drawing instruction, you “cancel out” this matrix transformation. That’s why the cursor sprite ends up exactly at the mouse position again: you first call ScreenToWorld, followed by ‘WorldToScreen’ (the matrix transformation), so the final result is exactly the screen position you started out with.

Now, drawing the mouse position was just a toy example, to show that the ScreenToWorld method worked well. For the Jewel Jam game, you can remove this cursor sprite again, and simply show the regular mouse pointer.

Later in this part of the book, you’ll use ScreenToWorld for something more interesting: there will be a “Help” button in the corner of the screen, but this button has been scaled to world coordinates as well. Therefore, to check if the mouse pointer lies on this button, you first need to translate the mouse position to world coordinates.

In other words: even though you may not see it yet, this translation between screen and world coordinates will become very useful later on!

12.5 What You Have Learned

In this chapter, you have learned:

- how to run a game in a window of a custom size;
- how to run a game in full-screen mode and how to let the player switch this on or off;
- how to scale the game world so that it preserves its aspect ratio, using a viewport;
- the difference between screen coordinates and world coordinates and how you can compensate for that (e.g., when the mouse is involved).

12.6 Exercises

1. *Coordinate Conversion*

What is the difference between *screen* coordinates and *world* coordinates? When do you need to convert between these two types? Try to give an explanation in your own words.

2. * *Side-Scrolling Games*

Many 2D games have a *side-scrolling* component to them. In side-scrolling games, a camera follows the player’s character as it moves through the game world. You see this a lot in *platform games* where the player needs to run and jump his/her way to a goal.

When side-scrolling is involved, the game world is no longer scaled to fit on a single screen; instead, the camera always shows a certain *part* of the game world. This makes the conversion between screen and world coordinates more complicated.

- a. Assume that the game has another member variable, `Vector2 cameraPosition`, which indicates the top-left corner of the camera in *world coordinates*. How would you need to change your `ScreenToWorld` method to take this into account?
- b. Now assume that the camera can also zoom in and out. Instead of a single position, there's now a `Rectangle` member variable that describes the area of the game world that is currently shown on the screen. How would you need to change your `ScreenToWorld` method now?

By the way, you won't have to create any side-scrolling games yet in this book. However, at the end of Chap. 26, you'll find an open *exercise* that challenges you to turn our platform game (*Tick Tick*) into a side-scrolling game.

Chapter 13

Arrays and Collections



The Jewel Jam game features a grid of jewels. All of these jewels are objects in the game world. Up until now, you’ve seen how you can add each game object as a separate member variable. In this chapter, you’ll learn about *arrays* and *collections*: special structures in C# that you can fill with many objects. That way, you don’t have to create a separate member variable for each jewel. You can probably imagine that arrays and collections are very powerful programming tools!

This chapter will start by discussing arrays and grids, both of which will be very useful in Jewel Jam. After that, we’ll talk a bit more about strings, because strings are (in some ways) very similar to arrays. At the end of the chapter, we’ll discuss collections in C#, which you could see as modern and more flexible versions of arrays. You will use collections more often in later chapters. However, “old-fashioned” arrays are still very useful for representing a grid of (game) objects.

At the end of this chapter, you’ll have a version of Jewel Jam in which each grid cell shows a random jewel. By pressing the spacebar, the player can move all jewels one row down, and the top row will be refilled with new random jewels.

13.1 Arrays: Storing a Sequence of Objects

In simple games (and other programs), it’s possible to use a separate member variable for each object. For instance, the Painter game contained only three paint cans, one cannon, and one ball. The Jewel Jam game, however, will contain a grid of 5×10 jewels. Let’s say that we have a Jewel class that describes the data and behavior of a jewel. It would be pretty unfortunate if you had to create a member variable for each jewel in the grid:

```
Jewel jewel1, jewel2, jewel3, jewel4, jewel5, ...
```

Luckily, there are better ways to deal with many objects in a structure such as a grid.

13.1.1 Basic Usage of Arrays

One very useful structure in C# is the **array**. An array represents a list of objects of a particular type. The syntax of arrays involves square brackets: [and]. For example, you can *declare* an array of integers as follows:

```
int[] myArray;
```

Here, the square brackets indicate that we're not declaring a single integer variable but an *array* of **int** variables. If you want to use this array, then declaring it is not yet enough. You also have to *initialize* it and specify how many elements your array will contain. For example, the following instruction will give `myArray` room for six integers:

```
myArray = new int[6];
```

This will reserve space in memory for six values of type **int**. You can then assign values to the array's items as follows:

```
myArray[0] = 4;  
myArray[1] = 8;  
myArray[2] = 15;  
myArray[3] = 16;  
myArray[4] = 23;  
myArray[5] = 42;
```

Watch out! In arrays (and pretty much everywhere else in computer science), *we start counting at zero*. That is, `myArray[0]` indicates the first element of the array, and `myArray[1]` indicates the second element. Likewise, `myArray[5]` indicates the *last* element, even though the length of the array is 6. This often leads to confusion among beginning programmers, so be careful.

In general, you can use `myArray[i]` to refer to the value at the *i*'th position in the array. This position is also called the **array index** (or simply *index* for short).

If you try to access an array index that doesn't exist (such as -1 or a number larger than the largest valid index), your program will crash. The compiler cannot warn you about this in advance!

You can also use the `myArray[i]` notation to *get* the value at a particular array index. This will give you an expression (in this case of type **int**) that you can use in a larger expression. For example, the following instruction is valid:

```
int x = myArray[3] + 12;
```

and it will store a value of 28 in the new local variable `x`. (After all, `myArray[3]` was set to 16 earlier.)

Just like with regular variables, existing values in the array will be overwritten as soon as you assign a new value:

```
myArray[3] = 100; // this will overwrite the old value of 16
```

Arrays also have a useful property named `Length`, which returns the number of elements in the array. In the current example, `myArray.Length` will return a value of 6. Note that this number is the full length of the array; it does not check how many values in the array have actually been *filled in*. So even if you've only assign a value to, say, `myArray[2]`, the expression `myArray.Length` will still have the value 6, because you promised earlier that this array can contain six numbers.

The `Length` property is read-only: you can only use it to *get* the array's length and not to *change* it. So the following instruction is *not* allowed:

```
myArray.Length = 108; // oops!
```



Quick Reference: Arrays

The following instruction:

```
double[] myArray = myArray[10];
```

declares and initializes an *array*, which is a list of values of a particular data type. In this case, `myArray` is a list of 10 **doubles**. Of course, you can use other names, lengths, and data types as well.

If `i` is an expression of type **int**, then the expression `myArray[i]` refers to the array element at position (“index”) `i`. You can use that to get or set values in the array:

```
myArray[6] = 3.141592;  
double myValue = myArray[6] + 1; // After this, myValue will store 4.141592.
```

The first element always has index 0, and the last element has an index that is one smaller than the array’s length (so in this case: 9).

The `Length` property returns the length of the array:

```
myArray[7] = myArray.Length + 0.5; // After this, myArray[7] will store 10.5.
```

13.1.2 Arrays in Memory

The representation of an array in memory is very similar to the representation of “normal” objects, which you’ve seen in Chap. 8. After the following declaration:

```
int[] numbers;
```

the memory will contain a new variable named `numbers`, ready to store a *reference* towards an actual array, which will be stored somewhere else in the memory. So, the variable initially has the value **null**, until you initialize it:

```
numbers = new int[5];
```

After this instruction, the memory will contain a block of five **int** values, and the `numbers` variable will point to that block. (Actually, the array’s memory block will be slightly larger than five integers, because it also needs some space to store the *length* of the array.)

You can also overwrite an entire array:

```
numbers = new int[108];
```

In that case, the `numbers` variable will start pointing to a *new* memory block (with room for 108 integers). And just like with objects, the memory that was reserved for the *old* array will be cleaned up, if there are no other variables pointing to it.

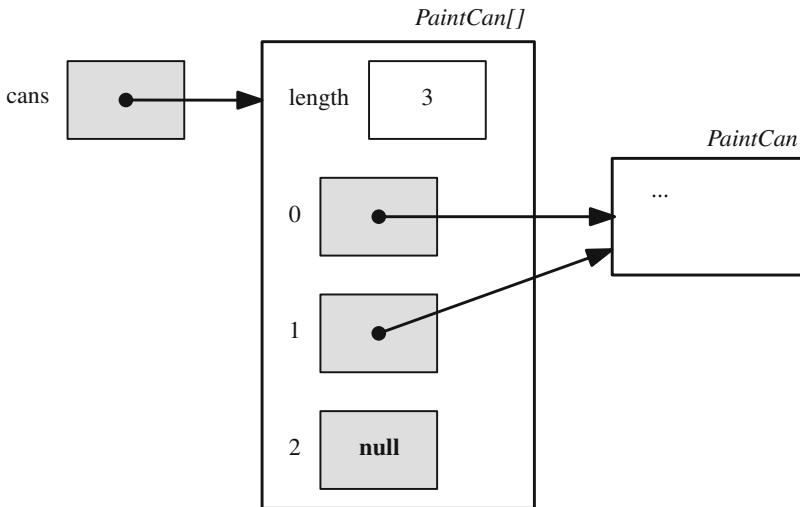


Fig. 13.1 The memory occupied by an example array of `PaintCan` objects

An array can store values of *any data type*. For example, you can use primitive types, struct types, class types, and even arrays again (but more about that soon). For primitive types and struct types, each array element will be stored directly inside the array's memory block.

For class types, each element in the array's memory block will store a *reference* to an actual object. In the Painter game, the following instruction:

```
PaintCan[] cans = new PaintCan[3];
```

would create an array with three references to `PaintCan` objects. Each of these references will initially have the special value **null**, until you let them point to actual instances of `PaintCan`. For example, imagine that we execute the following instructions for the `cans` array:

```
cans[0] = new PaintCan(...);
cans[1] = cans[0];
```

After these instructions, `cans[0]` points to a new `PaintCan` instance, `cans[1]` points to that same instance, and `cans[2]` still contains the value **null**. Figure 13.1 shows what the memory looks like after these instructions.

In general, if an element of an array has not been initialized yet, then it will store a *default value*. This works the same as with member variables: the default value is zero for numeric types, **false** for booleans, **null** for class types, and so on.

As a quick warning, you should be aware that computers and consoles don't have an infinite amount of memory. While it's perfectly allowed (by the compiler) to create an array with tens of thousands of values, you need to keep in mind that all of these values will be stored in memory somewhere. This is probably not much of a problem with simple data types such as **int**, but it can get out of hand if you store more complicated objects.

13.1.3 Combining Arrays with Loops

Because the indices of an array are nicely numbered from 0 onward, you will often see arrays being combined with **for** loops. For instance, the following code applies a value of 9 to each element in the array `myArray`:

```
for (int i = 0; i < myArray.Length; i++)  
    myArray[i] = 9;
```

Note that the stop condition here is `i < myArray.Length`. After all, the loop has to stop as soon as `i` has become equal to the array length, because your program will crash if you try to access `myArray[myArray.Length]`. For beginning programmers, it's tempting to start the **for** loop at index 1, like this:

```
for (int i = 1; i <= myArray.Length; i++) ...
```

This is dangerous if you try to use `i` immediately as an array index: you'll then skip the element at index 0, and your program will probably crash at the end of the loop. If you compensate for this somehow, then it's OK again:

```
for (int i = 1; i <= myArray.Length; i++)  
    myArray[i-1] = 9;
```

but it's usually less confusing to let your counter simply start at zero.

13.1.4 Multidimensional Arrays

The arrays you've seen so far represent a single list of values, but you can also create **multidimensional arrays**. For example, a two-dimensional array is very useful for representing a *grid* of objects. The following instruction declares a 2D array of integers:

```
int[,] myGrid;
```

and the following instruction initializes it to an array with 5 by 3 elements:

```
myGrid = new int[5,3];
```

You could think of a 2D array as a "table" of values, as opposed to a 1D array that represents only one "row" of values. Again, you need to specify the size of your array when you initialize it. In the same way, you can also create 3D arrays, and 4D arrays, and so on:

```
Color[,] rubiksCube = new Color[3,3,3];  
PaintCan[,,,] manyCans = new PaintCan[5,10,3,18];
```

but you probably won't need this very often. However, 2D arrays are useful for game programming, because many games (such as puzzle games) are based on a grid.

To get or set the value at a particular place in a 2D array, you need to specify two indices:

```
myGrid[0,0] = 2; // After this, myGrid[0,0] will store the value 2.  
myGrid[0,1] = myGrid[0,0] + 1; // After this, myGrid[0,1] will store 3.
```

Figure 13.2 shows what the memory looks like after these instructions.

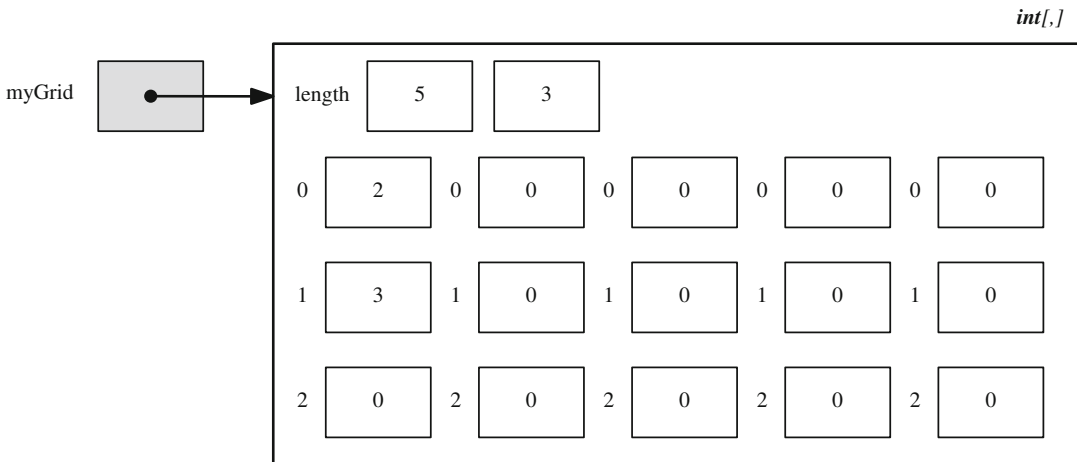


Fig. 13.2 The memory occupied by an example 2D array of integers

The `Length` property is available for multidimensional arrays as well:

```
int length = myGrid.Length;
```

For multidimensional arrays, `Length` will return the *total number* of elements. So, in this case, `myGrid.Length` will give a result of 15 (namely, 3 times 5). To access the length of a single dimension (in this case: the width or height), multidimensional arrays have a method named `GetLength`:

```
int width = myGrid.GetLength(0); // After this, width will store 5.
int height = myGrid.GetLength(1); // After this, height will store 3.
```

In this example, we store the first dimension of `myGrid` in a variable named `width` and the second dimension in a variable named `height`. If the 2D array indeed represents a grid of (game) objects, it's common to interpret the first dimension as the width of the grid and the second dimension as the height of the grid. However, this is not *required*: you can decide for yourself what the indices of your array “mean” in practice. In Fig. 13.2, we could also have drawn the array as a table with five rows and three columns. This is a matter of taste. For grid-based games, though, it seems to be common to interpret the two array dimensions in the way we're doing here.

2D arrays are often combined with *nested loops*. For instance, the following code assigns a value of 42 to each element of `myGrid`:

```
for (int x=0; x<myGrid.GetLength(0); x++)
{
    for (int y=0; y<myGrid.GetLength(1); y++)
    {
        myGrid[x,y] = 42;
    }
}
```

The curly braces can be left out here again, because the inner loop contains only one instruction, and the outer loop will treat the inner loop as one big instruction as well. For more information about nested loops, feel free to go back to Chap. 9.

In this last example, we've given our array counters the names `x` and `y`, to make clear that the array indices can be interpreted as *x*- and *y*-coordinates. But if we would (instead) interpret the array's first

dimension as the height and the second dimension as the width, then it would make more sense to reverse the names *x* and *y* as well.

It's pretty easy to get confused about code that involves arrays and indices. Remember: you can make your program as understandable as possible by giving your variables logical names. In this case, the variable names *x* and *y* make much more sense than, say, the names *wouter* and *arjan*.¹

13.1.5 Arrays of Arrays

An array can contain values of any type, but an array is also a type by itself. This means that you can also place arrays inside arrays! To create such an “array of arrays,” you use two pairs of square brackets. For example, the following instruction creates an array that can contain ten arrays of integers:

```
int[ ][ ] myArrays = new int[10][ ];
```

The main difference with a 2D array is that each of the “sub-arrays” can now have a different length. As you can see, you only need to specify the length of the “main” array at first. After that, you can initialize each *element* in the main array differently. For instance, you can do the following:

```
myArrays[0] = new int[200];  
myArrays[1] = new int[5];
```

and so on. Of course, you can repeat this “array-in-array” pattern as often as you want. The following instruction is also allowed:

```
int[ ][ ][ ][ ] moreArrays = new int[5][ ][ ][ ];
```

but we hope that you won't need it anytime soon.

For grid-based games, we recommend using 2D arrays (such as `int[5,10]`) instead of arrays in arrays. They're easier to work with, especially if your grid is a perfect rectangle.

13.1.6 Shorthand for Initializing an Array

One last thing: if you want to initialize an array and you already know exactly which elements it contains, then C# offers you a shorter way of writing that down. Take a look at the following line of code:

```
int[] anotherArray = { 4, 8, 15, 16, 23, 42 };
```

This instruction will initialize an array of six elements, filled in with these specific numbers. This single line is equivalent to first creating the array and then filling it in step by step.

You can also do this for multidimensional arrays, as follows:

```
int[,] bingoCard = {{1, 3, 7}, {12, 18, 23}, {33, 39, 45}};
```

¹Those are great names for *people*, though.

As you can see, you need to use a separate {...} expression for each row or column of the array, and they are grouped inside one large {...} expression.

After this example initialization, the expression `bingoCard[0,1]` will return the value 3. Here, the number 0 indicates the “sub-array” to look in (in this case, the array `[3, 71, 3, 7]`), and 1 indicates the index within that sub-array.

This might be the exact opposite of what you were expecting! We’ve said previously that the first index is often interpreted as the *x*-coordinate and the second index as the *y*-coordinate. If you read the numbers of `bingoCard` from left to right and interpret the three parts as the rows of your bingo card, then you’d expect that the number 3 lies at position (1, 0), and not at position (0, 1). So, if you want to use this short notation to initialize a 2D array, you need to watch out a bit. The compiler may have a slightly confusing idea of what the “rows” and “columns” are in a 2D array.

13.2 Putting It to Practice: Jewels in a Grid

Let’s apply your new knowledge of arrays to implement Jewel Jam’s grid of jewels. The game should feature a grid of 5×10 jewels, and we’re going to represent it by a 2D array with dimensions 5 and 10.

Let’s start with a simplified version of the game where there are only three possible jewels: a green diamond, a blue ellipse, and a red sphere. Include the three appropriate sprites for this: *spr_single_jewel1.png*, *spr_single_jewel2.png*, and *spr_single_jewel3.png*. You don’t have to create member variables for them yet; just including them via the Pipeline Tool is enough for now.

Eventually, the player will see a grid of sprites. However, for now, the game world will (internally) represent the jewels by *integers*. A value of 0 corresponds to the first jewel, a value of 1 corresponds to the second jewel, and a value of 2 corresponds to the third jewel. For each value in the grid, the Draw method will draw the appropriate sprite based on the number that we store in our 2D array.

13.2.1 Creating the Grid

Start by adding the following member variable to the JewelJam class:

```
int[,] grid;
```

This variable will store the array of “jewels” (represented by integers). Also add the following two member variables:

```
const int GridWidth = 5;
const int GridHeight = 10;
```

These are **constants** that we’re adding for convenience. They will represent the width and height of our grid. If you consistently write `GridWidth` and `GridHeight` everywhere (instead of a “hard-coded” 5 or 10), it will become very easy to give the game a different grid size if you ever want to do that. You’ll only have to change the definition of `GridWidth` or `GridHeight`, and the entire game will be adapted automatically! In general, it’s a good idea to use such constants as much as possible. It leads to clean code that’s easy to maintain.

Initializing Member Variables Immediately — You’ve just done something new: you’re now initializing member variables on the same line where you’re also declaring them. This is allowed, and it has (roughly) the same effect as initializing these variables in the constructor method. For member variables marked as **const**, it’s even *mandatory*: you always need to initialize a **const** variable immediately.

For non-**const** member variables, you can do the initialization either inside the constructor or immediately on the declaration line. It’s really a matter of taste. Personally, we prefer to let the constructor method initialize as much as possible.

At the end of the constructor of `JewelJam`, initialize the grid as follows:

```
grid = new int[GridWidth, GridHeight];
```

This will create a 2D array of 5 by 10 integers, and it will let the `grid` member variable point to that array. Because 0 is the default value of an **int**, the grid will automatically be filled with zeros.

13.2.2 Filling the Grid with Random Jewels

Now, let’s try to fill the grid with a random integer (0, 1, or 2) at each position. This will have the effect that the player sees a grid of random jewels. To do that, add a `Random` member variable just like in Chap. 9:

```
static Random random;
```

Initialize this variable in the constructor:

```
random = new Random();
```

You can then use `random.Next(3)` to get a random integer between 0 and 2. To fill the entire grid with random values, you can use a double **for** loop:

```
for (int x = 0; x < GridWidth; x++)
    for (int y = 0; y < GridHeight; y++)
        grid[x, y] = random.Next(3);
```

Add this code to your constructor as well.

13.2.3 Drawing the Jewels on the Screen

The `Draw` method should draw the grid of jewels on the screen, where each different integer value in the grid should lead to a different sprite being drawn. Let’s first declare (and initialize) two more member variables, which will serve as constants:

```
const int CellSize = 85;
Vector2 GridOffset = new Vector2(85, 150);
```

The `GridOffset` variable is the position of the top-left corner of the grid, in world coordinates. (We say “world coordinates” because we’re describing the *unscaled* version of the game world. The `Draw`

method might scale the game world to fit in a smaller screen, but that does not affect the game world itself.) The `CellSize` variable is the width and height of a grid cell, in world units. Again, the game world might get drawn in a scaled manner to fit on the screen. The cell size will be 85 pixels in the “unscaled” version of the game world.

Now go to the `Draw` method. After you draw the background sprite (but before calling `spriteBatch.End()`, you need to draw a sprite for each value in the grid. You can do this via a nested loop again:

```
for (int x = 0; x < GridWidth; x++)
{
    for (int y = 0; y < GridHeight; y++)
    {
        ...
    }
}
```

But what should you write at the position of “...”? Here, you need to draw a particular sprite depending on the value of `grid[x,y]`. Let’s first focus on the *position* of the sprite. The following instruction calculates this position:

```
Vector2 position = GridOffset + new Vector2(x,y) * CellSize;
```

This calculation makes sure that jewel (0, 0) will be drawn exactly at `GridOffset`. The other jewels will be offset by `CellSize` both horizontally and vertically, depending on their position in the grid. Take your time to understand how this instruction works exactly.

Now to *draw* an actual sprite at that calculated position. Hypothetically, let’s say that you have three separate `Texture2D` member variables: `jewel1`, `jewel2`, and `jewel3`. You could then write the following code:

```
if (grid[x,y] == 0)
    spriteBatch.Draw(jewel1, position, Color.White);
else if (grid[x,y] == 1)
    spriteBatch.Draw(jewel2, position, Color.White);
else
    spriteBatch.Draw(jewel3, position, Color.White);
```

This works, but there’s also a nicer way to do it. The trick is to store the three `Texture2D` objects inside an *array* as well and not as three separate member variables. Give the `JewelJam` class another member variable, which will store an array of sprites:

```
Texture2D[] jewels;
```

In the `LoadContent` method, initialize this array so that it can store three elements:

```
jewels = new Texture2D[3];
```

and then load the three sprites and store them at indices 0, 1, and 2 of your array:

```
jewels[0] = Content.Load<Texture2D>("spr_single_jewel1");
jewels[1] = Content.Load<Texture2D>("spr_single_jewel2");
jewels[2] = Content.Load<Texture2D>("spr_single_jewel3");
```

Do you see what’s going on here? Instead of using a member variable per jewel sprite, we’re storing all sprites together in one array. This way, drawing the appropriate sprite becomes very easy. In the `Draw` method, inside the nested loop, you can now simply write this:

```
int jewelIndex = grid[x, y];
spriteBatch.Draw(jewels[jewelIndex], position, Color.White);
```

The first instruction gets the number stored at the current grid element: this number will be 0, 1, or 2. Then, the expression `jewels[jewelIndex]` will exactly give you the first sprite if `jewelIndex` is 0, the second sprite if `jewelIndex` is 1, and the third sprite if `jewelIndex` is 2. Therefore, the second instruction will automatically draw the correct sprite.

You could also write these instructions in one line:

```
spriteBatch.Draw(jewels[grid[x, y]], position, Color.White);
```

This is perfectly valid code, but we can imagine that you find these nested square brackets a bit confusing to read. You can also keep using the two-line version from before.

This is another example of how powerful arrays can be. If you're ever going to support more than three types of jewels, you don't have to write more **if** and **else** instructions. Instead, you can simply add another sprite to the array, and everything will work automatically.

Compile and run the game, and you should see a grid filled with random jewels. Each time you start the game, the grid will be filled differently.

13.2.4 Letting the Player Change the Array

As another example of what you can do with arrays, we're going to add some player interaction. When the player presses the spacebar, we want all jewels to move one row down. The top row of the grid should then be refilled with new random jewels. (This functionality will not appear in the final JewelJam game, but it's a good exercise for now.)

How should you implement this? You can start by checking for a key press in the Update method:

```
if (inputHelper.KeyPressed(Keys.Space))
    MoveRowsDown();
```

where `MoveRowsDown` is a method that you still need to implement. This method has a **void** return type, because it doesn't return anything; it only updates the grid member variable in a particular way. Go ahead and create the header of this method, followed by an empty body.

To move all jewels one row down, you'll have to use yet another nested loop. Basically, for each cell of the grid, you'll have to fill that cell with the value that's currently stored in the cell directly *above* it. It's a good idea to do this one row at a time. This means that you should *first* iterate over the grid's rows and *then* (inside that loop) iterate over the elements in a row. You can ignore the first row, because you don't have to replace anything there yet. So, it makes sense to start out as follows:

```
for (int y = 1; y < GridHeight; y++)
    for (int x = 0; x < GridWidth; x++)
        ...
```

Notice the start condition `y = 1` in the outer loop: we don't want to do anything for the top row, so we can ignore the row with index 0. At "...," you could try to write the following:

```
grid[x, y] = grid[x, y-1];
```

This will fill the cell (x, y) with the value that is currently stored in the cell above it. Compile and run the program now, and press the spacebar. You'll see that something very strange happens: all rows will suddenly look the same! That's not what we had in mind.

If you look at the loop more carefully, you'll understand why this happens. This code first fills row 1 with the jewels from row 0. It then fills row 2 with the jewels from row 1. But by that time, rows

0 and 1 are already the same. Therefore, row 2 will *also* contain the jewels from row 0! The code continues like this for all rows. At the end, all rows of the grid will contain the same values.

Apparently, this code processes the rows in the wrong order. If you instead iterate over the rows *from bottom to top*, you will not have this problem. So, change the outer **for** loop as follows:

```
for (int y = GridHeight - 1; y > 0; y--)
```

This way, the program will start by filling in the bottom row (at `GridHeight - 1`) with the values from the second-to-bottom row (at `GridHeight - 2`). After that, it will fill the second-to-bottom row with the values from the third-to-bottom-row and so on. Do you see how this avoids the “repeated copying” of our previous attempt? (Also, note that the loop will still ignore the top row, due to the loop condition `y > 0`.)

Leave the rest of the code the same, and compile and run the program again. Now, pressing the spacebar will correctly move all jewels one row down.

After moving all rows down, you need to do one last thing: refill the top row with new random jewels. You can do that as follows, after the entire double **for** loop has finished:

```
for (int x = 0; x < GridWidth; x++)  
    grid[x,0] = random.Next(3);
```

This concludes the `MoveRowsDown` method.



CHECKPOINT: JewelJam2

As you’ve hopefully noticed by now, arrays allow you to do very powerful things, but they can also be quite confusing to work with. Especially when **for** loops and indices are involved, it’s easy to make a small mistake. Mixing up 0 and 1, or `<` and `<=`, can cause your program to behave unexpectedly or even to crash. There’s no “holy grail” for avoiding this confusion. You’ll simply have to do a lot of programming and get a feeling for how it all works. And, if it’s of any consolation, even the most experienced programmers still make these mistakes every now and then.

13.3 Strings and Arrays

In the rest of this chapter, you’ll learn about the similarity between arrays and *strings*. It turns out that the `String` class of C# has a lot in common with arrays. You will not immediately use this topic for the Jewel Jam game, but this chapter is still the best place for taking a closer look at it.

If you think about it, a piece of text is really not much more than a sequence of characters. Therefore, an object of type **string** is very similar to an object of type **char[]**. For example, you can access a particular character in a string using “array-like” square brackets:

```
string txt = "Hello World!";  
char c = txt[0]; // After this instruction, c will store the character 'H'
```

As you can see, strings also use an index of 0 to indicate the first character. This means that the expression `txt[txt.Length-1]` will give you the *last* character and that the expression `txt[txt.Length]` will give an error that can make your program crash.

Because of this similarity between strings and arrays, you can also use a **for** loop to iterate over the characters in a string:

```
for (int i = 0; i < txt.Length; i++)
{
    // do something with char[i];
}
```

You can use such a loop to do many interesting things, such as finding a certain character in a string, checking if two strings are equal, converting a string to uppercase or lowercase, and so on.

Strings and char Arrays — Strings and characters exist in many different programming languages. However, the type **string** has slightly different details in each programming language, whereas a **char** array is something that many languages treat in exactly the same way.

In older programming languages such as C, there isn't even a separate data type for strings, and **char** arrays are the only way to represent pieces of text. Therefore, newer languages such as C# and C++ contain methods for converting a **string** to a **char[]** (or vice versa). This is useful if your program ever has to communicate with another program written in a different programming language.

13.3.1 Immutability

There's one very important difference between strings and regular arrays: strings are **immutable**. This means that you can't change a string anymore after you've created it. One consequence of this is that you can't use the bracket notation to change the characters in a string. So, the following instruction is *not allowed*:

```
txt[0] = 'J';
```

because this would change the contents of the string `txt`, which is forbidden.²

The `String` class contains many methods that may *look like* they change a string. We'll discuss these methods next. However, it's worth knowing that these methods will actually return a *new* string instance, without changing the initial string at all. This is (again) because strings are immutable: once you've created a string, you can't change it anymore. The creators of the `String` class have made this choice on purpose, because it can be confusing if strings suddenly receive different content due to the "side effects" of a method.

²Too bad: *Jello World* sounded pretty good!

13.3.2 Useful String Methods

The `String` class already contains several useful methods for string-related tasks that are commonly used. Here are a few examples:

- **string** `Substring(int startIndex, int length)`: selects a part of the string indicated by two positions and returns it as a result. For example, with the `txt` variable from before, the expression `txt.Substring(1, 4)` will return the string "ello" (a string of four characters long, starting at index 1).
- **string** `Concat(object s)`: glues a second string behind the string and returns that concatenation as a result. If the parameter is something other than a string, the method `ToString` will be called on the object first. All data types have a `ToString` method by default. If you want, you can let your own classes override this method, so that your objects will look different when converted to a string.
- **bool** `Equals(string s)`: compares the string character-by-character to another string, and returns **true** if all characters are the same (and **false** otherwise);
- **int** `IndexOf(string s)`: determines at which spot the substring `s` appears in the current string for the first time;
- **string** `Insert(int p, string s)`: inserts an extra string `s` into the current string at position `p` and returns the result as a new string.

You can use these methods (and many more) for free in your C# programs. Whenever you want to do something special with a string, it's worth checking if the `String` class doesn't already contain a method that does exactly what you want. In the exercises of this chapter, you will write some of these methods yourself, to get a feeling of what is happening "in the background" if you call such a method.

13.3.3 String Operators

The `String` class also overrides several **operators**. You've already seen the `+` operator, which you can use to glue strings together. In fact, the `+` operator does exactly the same as the `Concat` method we just mentioned. There's also a `+=` variant of this operator. Given a string `txt`, the following instruction:

```
txt += "How are you?";
```

is a shorter version of this instruction:

```
txt = txt + "How are you?";
```

Furthermore, the `String` class overrides the `==` operator, so that it does the same as the `Equals` method. In other words, the `==` operator will return **true** if two strings contain exactly the same text. In several other languages (such as Java), the `==` operator does *not* have this special behavior for strings. There, `==` will check if two **string** variables point to the same *object in memory*, so it can return **false** even if two strings look the same to us humans. If you ever start to work in a different programming language than C#, this is a difference to watch out for.

13.4 Collections

A downside of arrays is that you need to specify the *length* of an array in advance. This can be a problem if you want to use an array to represent the list of game objects in your game world. What if you want to spawn another enemy and *add* it to the list? Or what if an item has been picked up and should be *removed* from the list? These things are difficult to do with a regular array.

To make up for this, C# also contains a concept called **collections**. A collection is a more advanced container for a set of objects. There are many different collections in C#, each with their own extra functionality (compared to arrays) and with their own advantages and disadvantages.

Collections are defined in a separate C# library. If you want to use collections, you'll have to include that library at the top of your file:

```
using System.Collections.Generic;
```

In this section, we'll discuss the most commonly used collection (the `List` class), and we'll talk about a few related topics.

13.4.1 The *List* Class

One collection that you'll use very often is the `List` class. A `List` has many of the same possibilities as an array: for example, you can use square brackets to get or set the element at a certain index. There is also a property that tells you how many elements the list has in total: this property is named `Count` (whereas for arrays, it's named `Length`). The main advantages of a `List` is that it allows you to add and remove elements during the program.

If you create a *List* (or another type of collection), you need to use angular brackets (< and >) to indicate what type of object your list contains. For example, to create a list of integers, you can write the following:

```
List<int> myIntegers = new List<int>();
```

As you can see, a list is a “full-fledged” object that you have to create via the keyword **new**, just like an array. To create a list with another type of objects, you simply replace `int` by the data type that you want:

```
List<Cannon> myCannons = new List<Cannon>();
```

When you create a new list, it's still completely empty at first. To add new objects to it, you can use the `Add` method:

```
myIntegers.Add(1);  
myIntegers.Add(42);  
myCannons.Add(new Cannon());
```

The `Add` method will add your object at the *end* of the list. To add an object at a different position, you can use the `Insert` method instead. This method has a second parameter that indicates the index at which you want to insert the object. For instance, the following instruction:

```
myIntegers.Insert(9001, 0);
```

will add the number 9001 at the beginning of `myIntegers`. The other elements (1 and 42) will both move one position further in the list.

Note: if your list is very long, then adding an object near the beginning (with `Insert`) takes more time than adding the object at the end (with `Add`), because the `Insert` call will cause the other list elements to move around. You probably won't notice the difference very quickly, but in larger and more complicated programs, it's something to keep in mind.

The opposite method of `Insert` is called `Remove`. You can use this method to remove an object at a particular index. Any objects *after* that index will move one position to the front. So, after the following instruction:

```
myIntegers.RemoveAt(0);
```

the `myIntegers` list will consist of the numbers 1 and 42 again. Just like with `Insert`, it's good to be aware of the extra work that these methods do in the background.

Just like the `String` class, the `List` class contains many other useful methods for tasks that are commonly used. For example, there are methods for finding a certain object in a list, for clearing a list entirely, for reversing the order of all elements, and so on. We won't go into the details of all of these methods—you can find lots of information about it in the documentation `List` class itself, as well as on the Internet. You'll see some examples throughout this book when you need them.

You will use the `List` class in the next chapter, when you're going to create a nicer way to structure the game world. You don't need this class yet in *this* chapter, though.

13.4.2 Lists, Loops, and the *foreach* Keyword

Of course, you can combine lists with loops again. For example, the following code will keep adding cannons to the `myCannons` list until it has 500 elements:

```
while (myCannons.Count < 500)
    myCannons.Add(new Cannon());
```

And the following code will compute the sum of all numbers in the `myIntegers` list:

```
int result = 0;
for (int i = 0; i < myIntegers.Count; i++)
    result += myIntegers[i];
```

Sometimes, you'll want to loop over the elements in a list *without* having to know (during a single iteration of the loop) where you currently are inside that list. For that purpose, C# offers a special type of loop that doesn't use an explicit counter variable. This special loop uses the keyword **foreach**. For example, the following code is an alternative way of computing the sum of all elements `myIntegers`:

```
int result = 0;
foreach (int number in myIntegers)
    result += number;
```

This **foreach** loop will go over the elements of `myIntegers` from beginning to end. In each iteration of the loop, the variable `number` will refer to the element you're currently looking at. In the background, there is a hidden **enumerator** (sometimes also called an *iterator*) that points at the current element. After each iteration of the loop, the enumerator will move one position further in the list. The **foreach** loop will automatically stop when the enumerator has reached the end of the list.

The **foreach** keyword can be used for *other collections* as well, and even for regular *arrays*. Using **foreach** instead of **for** can lead to even shorter code that is easier to read.

However, be aware that the lack of a counter (such as `int i`) means that you're not keeping track of the current *index* anymore. In the Jewel Jam game, we're using a 2D array to represent the grid of jewels. To draw these jewels at the correct position on the screen, you *need* to use regular **for** loops (and not a **foreach** loop), because the position of each sprite should depend on the current index in the loop.



Quick Reference: The `foreach` Instruction

Given an array or collection, such as the following:

```
List<int> myList;
```

the following code loops over all elements in the list:

```
foreach (int myInt in myList)
{
    ...
}
```

In each iteration of the loop, `myInt` is a local variable that refers to the current object in the list. The program will automatically make sure that `myInt` refers to the next object after each iteration and that the loop finishes after you've visited all elements.

Of course, the data type in the loop's header (in this case `int`) should match the type of object in your array or collection.

13.4.3 Other Collections

There are other useful collections in the `System.Collections.Generic` library. Basically, these are all types of containers, each with certain advantages and disadvantages.

For example, the `Queue` class represents a list of objects where you can only *add* elements at one end and *remove* elements at the other end. You could compare this to a queue of people in a supermarket. The `Stack` class represents a list of objects where you can only add and remove elements at one end (the "top" of the list). You could compare this to a stack of bricks where you can only pick up the top brick or add another brick to the stack.

In both of these classes, it's more difficult to retrieve an element at a certain *index*, because these classes don't really keep track of where all objects are. They only focus on a few very specific tasks, and they make sure that you can do these tasks very efficiently.

Another interesting collection is the `Dictionary` class. You can use a `Dictionary` to organize your objects in such a way that you can quickly search for any object. A dictionary is based on *two* data types instead of one. For example, the following instruction:

```
Dictionary<string,Cannon> cannons = new Dictionary<string,Cannon>();
```

creates a new dictionary that can store `Cannon` objects, which will be *organized* according to `string` values. Each item in the dictionary will be a combination of a `string` and a `Cannon`. In general, such a dictionary item is called a *key-value pair*. To add an item to the dictionary, you can write the following:

```
cannons["Charlie"] = new Cannon();
```

This will create a new `Cannon` instance and store it in the dictionary using the key "Charlie". After this instruction, the expression `cannons["Charlie"]` will give you the `Cannon` instance that you've just added.

Behind the scenes, the Dictionary class makes sure that you can quickly find the value that belongs to a particular key. Note that each key can only be used once. If you assign a new value to an existing key, you will overwrite the old value.

There's much more to say about dictionaries, but we'll save this for a later point in the book. Of course, if you're curious, you can already look for more information yourself.

There are many more kinds of collections, but we cannot possibly treat them all here. Which collection(s) should you use for which application? This is an important question that programmers should often ask themselves. However, to really understand the ins and outs of each type of collection, you'll have to know more about *data structures* and *algorithms*. But that's pretty advanced material: for example, if you study computer science at a university, you'll typically have separate courses about these topics. Treating those topics in detail would be way too much for this book.

For now, we'll leave it at this. In the later chapters of this book, you'll use a few collections (such as List and Dictionary) to build your games, but we won't go into the details of why one collection would be better than another.

Generics — The notation `List<int>` indicates a list of integers, and `List<PaintCan>` indicates a list of `PaintCan` instances, and so on. It looks like C#'s definition of the List class “leaves open” what type of object you put in the list. That way, you can use the List class for any type of object that you want, simply by writing that type between the `<` and `>` symbols.

A class that uses `<` and `>` in this way is called a **generic class**. This explains why the library that contains the List class is called `System.Collections.Generic`.

Generic classes are very powerful. You could see this concept as an “extended version” of inheritance: instead of having to write a class `PaintCanList` that inherits from List, you can simply write `List<PaintCan>` and everything will work automatically. It's useful to think of the `<` and `>` symbols as the word “of”: for example, a `List<PaintCan>` object represents “a list *of* paint cans.”

You can write generic classes yourself, too. In a class header, simply write `<T>` immediately after the class name, and then `T` becomes a “generic type” that users of your class can fill in later. (You can also use other letters than `T`, or even entire words, but `T` seems to be a common choice among programmers.)

You could also write generic *methods* that “leave open” what type of parameters they use. Overall, programmers use the term **generics** to refer to generic classes and methods.

However, you won't have to write your own generics in this book yet, so we won't talk about it further. The difference between inheritance and generics is very subtle. For beginning programmers, it's easy to make the wrong choice, for example, by writing a generic class where “regular” inheritance would have been much more logical. In our opinion, it's better to save this topic for later in your programming career.

13.5 What You Have Learned

In this chapter, you have learned:

- how to store multiple objects of the same type in an array;
- how to use a two-dimensional array to represent a grid of values;

- more about the **string** data type and how strings are very similar to arrays;
- how to use collections in C#, such as List and Dictionary;
- how to use the **foreach** instruction to traverse an array or collection and perform operations on its elements.

13.6 Exercises

1. *Lists*

Consider the following code:

```
List<int> numbers = new List<int>();
numbers.Add(5);
numbers.Add(4);
numbers.Add(3);
numbers.Add(2);
numbers.Add(1);
numbers.RemoveAt(0);
numbers.RemoveAt(1);
numbers.RemoveAt(2);
```

What does the list numbers look like at the end of these instructions? Go over the instructions one by one, and keep track of the contents of the list as you go along.

2. *Methods with Arrays*

This exercise lets you write methods that loop over arrays.

- Write a method `CountZeros` that has an array of integers as its parameter. The method should return the number of zeros in the array.
- Write a method `AddArrays` that has two integer arrays as its parameters. You may assume that both arrays have the same length. The method should return another array of that length, where each value is the sum of the values of the input arrays. So, for each position i , the result array should contain the sum of element i of the first array and element i of the second array. For example, given `int[] array1 = {0, 3, 8, -4}` and `int[] array2 = {10, 2, -8, 8}`, the result of the method call `AddArrays(array1, array2)` will be another array with the contents `{10, 5, 0, 4}`.

3. *Uppercase and Lowercase*

In Chap. 11, you've seen a Boolean expression that checks whether a **char** value represents an uppercase letter. Let's apply this to entire strings.

- Write a method `IsAllUppercase` that takes a string as an argument and that returns whether or not this string consists completely of uppercase characters.
- Write a method `ToLowercase` that takes a string as an argument and that returns a new string in which all uppercase letters have been replaced by their lowercase version. For example, the method call `ToLowercase("HELLO World! :)")` should return the string `"hello world! :)"`. *Hint:* Loop over the characters in the input string, and build your result one character at a time. Be aware that not all characters are letters from the alphabet!

4. *More string Methods*

Actually, the `String` class already has many built-in methods such as `ToLower`. In this question, we'll look at a few other useful methods of the `String` class—but again, you will write them yourself.

- a. Write a method `FirstPosition` which has two parameters: a **string** and a **char**. The method should return the position in the string at which the **char** parameter first occurs. If the character doesn't occur in the string, the method should return `-1`. For example:
 - `FirstPosition("wouter", 'j')` should return `-1`.
 - `FirstPosition("arjan", 'j')` should return `2`.
 - `FirstPosition("jeroen", 'j')` should return `0`.
- b. Write a method `Replace` that takes three parameters: a string and two characters. This method should return a new string in which each occurrence of the first **char** has been replaced by the second **char**. For example:
 - `Replace("Good morning", 'o', 'u')` should return `"Guud murning"`.
 - `Replace("Choose your character", 'o', 'e')` should return `"Cheese yourcharacter"`.
 - `Replace("A+2+#?", '+', '9')` should return `"A929#?"`.
- c. Write a method `EndsWith` that takes two parameters of type **string**. This method should return whether or not the first string ends with the second string. For example:
 - `EndsWith("All your base are belong to us", "long to us")` should return **true**.
 - `EndsWith("Can you hear the echo?", "the echo?")` should return **true**.
 - `EndsWith("abcdefg", "abcdefghijk")` should return **false**.

In your implementation, you're allowed to use existing methods in the **string** class. For an extra challenge, try to create a version that does *not* use the `Substring` method.

5. Collections

Write a method `RemoveDuplicates` which receives as `List<int>` as its parameter. The method should remove all duplicate numbers in the given list. For example, if the list contains the numbers 0, 1, 3, 2, 1, 5, 2, this method should change the list so that it only contains 0, 1, 3, 2, 5. Watch out: this method should change the input list itself and *not* return a whole new list.

6. Searching in Arrays

In this exercise, you'll write several methods that each take an array of **doubles** as a parameter.

- a. Write a method `Largest` that returns the *largest value* that occurs in the array.
- b. Write a method `IndexLargest` that returns the *array index* of the largest value in the array. If this largest value occurs more than once in the array, the method should return the index of the first occurrence.
- c. Write a method `IndexOf` that takes two parameters: an array of **doubles** and another **double** value. This method should return the array index at which the given **double** value occurs for the first time. For example, given the following array:

```
double[] values = {9,12,9,7,12,7,8,25,7};
```

the method call `IndexOf(values, 12)` should return the value 1, because the number 12 first occurs at index 1 of the array.

- d. * Write a method `HowManySmallest` that returns *how often* the smallest value of the array occurs inside that array. For example, given the following array:

```
double[] values = {9,12,9,7,12,7,8,25,7};
```

the method call `HowManySmallest(values)` should return the value 3, because the smallest value (7) occurs three times in the array.

7. * *Sorting and Searching*

In this exercise, you'll use the solution of the previous exercise to do more advanced things. *Warning*: this is a difficult exercise! It's actually more related to data structures and algorithms, and not so much to programming.

- a. Write a method `Sort` that takes an array of **doubles** as input, and returns an array in which the values of the input array have been *sorted from smallest to largest*.

Hint: First find the largest value in the array, using `IndexLargest` from the previous exercise. Swap this value with the value at the end of the array. After doing that, the largest value is already at the end of the array where it belongs. Then, find the largest value in the rest of the array and swap that value with the second-to-last value in the array. Continue like this until the entire array has been sorted.

- b. If you know that an array is already sorted, it is possible to create a smarter version of the `IndexOf` method (for finding a certain value in an array).

Instead of going through the whole array from start to finish, you can now start by looking at the *middle element* of the array. If that element is smaller than the value you're looking for, you know for sure that the left half of the array cannot contain your value either (because the left half contains only smaller numbers). So you can continue by looking at the *right half* of the array. But if the middle element is larger than (or equal to) the value you're looking for, you need to look further at the *left half* of the array. You can continue like this (splitting the interesting part of the array in half) until you've found the correct value or until you're sure that the desired value does not occur in the array. Searching like this is much more efficient than going through the entire array (but it only works if the array is sorted!).

Write this improved `IndexOf` method. Use two integers to keep track of the boundaries of the piece of array that you're searching in.

Chapter 14

Game Objects in a Structure



In Chap. 13, you've seen how you can use arrays and lists to keep track of the objects in your game. In this chapter, you will use this idea (combined with inheritance) to start building your own "game engine." This will be a framework for the final Jewel Jam game and also for the other games you'll make later in this book.

At the end of this chapter, your game will still do the same, but it will (again) have a much better program structure. You will also have an even better understanding of lists, inheritance, and communication between objects in a large program. Last but not least, you'll have learned about a very useful programming concept: *recursion*.

14.1 Creating Your Own Game Class

You may have noticed that the JewelJam class contains quite a lot of code for setting up the game: preparing the graphics device, dealing with different screen sizes, handling basic player input for quitting the game, and so on. In this section, you'll put all of this "common code" into a separate class named ExtendedGame and then turn JewelJam into a subclass of ExtendedGame. The later games of this book will also be based on ExtendedGame again, so it's useful to prepare that class as soon as possible.

14.1.1 Outline, Member Variables, and Properties

Create a new class ExtendedGame and make sure that it extends the Game class of MonoGame. Give your class all member variables that JewelJam used to have, except for the variables that are really specific for the Jewel Jam game (namely, the sprites, the grid, and the grid size). Your new class should start out like this:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using System;

class ExtendedGame : Game
{
    // Standard MonoGame objects for graphics and sprites.
    protected GraphicsDeviceManager graphics;
    protected SpriteBatch spriteBatch;
```

```
// An object for handling keyboard and mouse input.
protected InputHelper inputHelper;

// The width and height of the game world, in game units.
protected Point worldSize;

// The width and height of the window, in pixels.
protected Point windowSize;

// A matrix used for scaling the game world so that it fits inside the window.
protected Matrix spriteScale;
}
```

Note: we've added the keyword **protected** to all of these variables, because we expect that subclasses of `ExtendedGame` will want to use them. You can also choose to leave this keyword out (thus making all member variables **private**) and marking a variable as **protected** as soon as you really need it. That's entirely up to you.

Next, give this class a static property that contains a random number generator:

```
public static Random Random { get; private set; }
```

That way, any class in your game can write `ExtendedGame.Random.Next(...)` to draw a random number. (Of course, you'll have to *initialize* this `Random` object somewhere, but we'll get to that soon.)

In the *Painter* game, it was a bit unfortunate that the constructor of each game object required a `ContentManager` object as a parameter. We constantly had to pass around the `ContentManager` instance of the main game class; otherwise we couldn't load any sprites. From now on, let's do things a bit differently. Just like with the `Random` object, we'll give `ExtendedGame` a static property that can be accessed from everywhere:

```
public static ContentManager ContentManager { get; private set; }
```

Whenever you need to load an asset, you can now write `ExtendedGame.ContentManager` instead of requiring a special method parameter.

Also give your class the `FullScreen` property by copying and pasting it from the `JewelJam` class. You'll have to change its access modifiers: make the entire property **public**, and make the **set** part **protected**. That way, everyone can ask whether the game is running in full-screen mode, but only `ExtendedGame` and its subclasses can *change* that value.

14.1.2 Methods

Next, add the methods related to the screen size: `ApplyResolutionSettings`, `CalculateViewport`, and `ScreenToWorld`. You can directly copy and paste these from your old `JewelJam` class. What remains is to fill in the more interesting methods: the constructor, `LoadContent`, and the methods related to the game loop.

In the constructor of `ExtendedGame`, you can set up some basic objects that don't require the graphics device yet. You can also choose a default window size and world size.

```
protected ExtendedGame()
{
    Content.RootDirectory = "Content";
    graphics = new GraphicsDeviceManager(this);
```

```

inputHelper = new InputHelper();
Random = new Random();

// default window and world size
windowSize = new Point(1024, 768);
worldSize = new Point(1024, 768);
}

```

If a child class of `ExtendedGame` wants to apply a *different* window or world size, it can overwrite this behavior in its own constructor method.

As usual, `LoadContent` is the first opportunity to do things with the graphics device. In this method, you can initialize the sprite batch and apply the resolution settings:

```

protected override void LoadContent()
{
    spriteBatch = new SpriteBatch(GraphicsDevice);

    // store a static reference to the ContentManager
    ContentManager = Content;

    // by default, we're not running in full-screen mode
    FullScreen = false;
}

```

In the `Update` method, you can at least update the `InputHandler` object and use it for handling the basic key presses. In the example below, we've created a separate method `HandleInput` for this, which will prove to be useful later on.

```

protected override void Update(GameTime gameTime)
{
    HandleInput();
}

protected virtual void HandleInput()
{
    inputHelper.Update();

    // quit the game when the player presses ESC
    if (inputHelper.KeyPressed(Keys.Escape))
        Exit();

    // toggle full-screen mode when the player presses F5
    if (inputHelper.KeyPressed(Keys.F5))
        FullScreen = !FullScreen;
}

```

Note that the `Update` method is marked as **override**, because it is overriding the standard `Update` method of the `Game` class. Also, both of these methods are marked as **protected**, because we expect that a child class may want to add its own (game-specific) behavior.

For the `Draw` method, there's not much to do yet. Apart from a few lines, the code in `Draw` has always been very game-specific so far. For now, don't add this method to `ExtendedGame` yet. We'll get back to this method later.

14.1.3 Turning JewelJam into a Subclass

You can now turn JewelJam into a subclass of ExtendedGame:

```
class JewelJam : ExtendedGame
```

Thanks to this inheritance, you can remove most of the code from JewelJam, because this class now inherits a lot from its parent class. Here's a quick outline of what the JewelJam class should contain. Try to work out the details yourself, or take a look at the JewelJam3a example project if you get stuck.

- All member variables related to sprites and the grid should remain. So, keep the following member variables in place: background, jewels, grid, GridWidth, GridHeight, CellSize, and GridOffset.
- All other member variables can go, because they're already in ExtendedGame now. The same goes for all properties and methods that you've moved entirely to ExtendedGame.
- The constructor should contain the line `IsMouseVisible = true`; (because ExtendedGame doesn't do that yet), and it should initialize the grid of random jewels. Note that you now need to write `Random` instead of `random`, because you're now using the *public property* `Random` instead of the old *private member variable* `random`. You don't have to change the *header* of the constructor: the compiler will automatically understand that the parent constructor should be called as well.
- In `LoadContent`, you first need to call `base.LoadContent()` to do all basic set-up tasks. You can then initialize background and jewels, just like before. Finally, you need to update the `worldSize` variable, because this game world will not have the default size of 1024×786 . Also, make sure to recalculate how the world should be scaled. You can do that by setting the `FullScreen` property again.
- The `Update` method should call `base.Update(gameTime)` (to execute the standard behavior of ExtendedGame), followed by the extra code that moves all jewels one row down when the spacebar is pressed.
- The `Draw` method can stay as it is.
- The `MoveRowsDown` method should stay as well, although you need to change `random` into `Random` again.

Once you've updated the class according to these guidelines, you can compile and run the game again. You should still be able to do exactly the same things as in the previous version. But from a programming point of view, a lot of the "dirty" work is now nicely hidden inside the ExtendedGame class. The JewelJam class can now really focus on the specific work for this game.



CHECKPOINT: JewelJam3a

14.2 A List of Game Objects

In this section, you'll work on another piece of code that you can reuse in all of your games: a list of game objects. Do you remember how the Painter game contained a background, three sprite cans, a cannon, and a ball? This was the representation of the game world in that game. In Jewel Jam, the game world will contain a grid of jewels, a moving mine cart, and a few more objects that we'll talk about later.

Although these two game worlds are very different, they have an important *similarity* as well: they both consist of *game objects* that each have their own way of updating and drawing themselves. This suggests that you could model every game world as a collection of game objects, and then let each object in that collection update and draw itself. That's exactly what we are going to do in this section, using the things you've learned about collections and inheritance.

14.2.1 The *GameObject* Class

First, let's create a class `GameObject` that will become the parent class of all objects in our game. You could see this as an even more “general” version of the `ThreeColorGameObject` class from `Painter`. In fact, let's make it so general that it doesn't even have to contain a *sprite*. You'll see later why this is a good idea.

With this in mind, what is the bare minimum that a game object should contain? Well, a game object at least has a position and a velocity (which might be zero). For now, we expect that the position is something that can be changed by other objects, while the velocity is not. According to us, it makes the most sense to turn the position into a public property and the velocity into a protected member variable. Other options are definitely possible, though.

There should be an `Update` method in which the object updates its position according to the velocity. The constructor method can initialize both vectors to `Vector2.Zero`. Furthermore, there should be `HandleInput` and `Draw` methods. These methods don't do anything by default, because each specific object should decide for itself what to do there. However, it's good to already promise that these methods exist, so that child classes can override them. Let's also include a `Reset` method, because we expect that our games (and the objects within them) will want to reset themselves, for example, when the player restarts the game.

In `Jewel Jam` and the games of this book, it will be useful to turn objects “on and off” by making them visible or invisible. To enable this, let's give `GameObject` a public property (`bool Visible`) that can be get and set.

Listing 14.1 The `GameObject` class in `JewelJam3b`

```

1  using Microsoft.Xna.Framework;
2  using Microsoft.Xna.Framework.Graphics;
3
4  class GameObject
5  {
6      public Vector2 Position { get; set; }
7      protected Vector2 velocity;
8
9      public bool Visible { get; set; }
10
11     public GameObject()
12     {
13         Position = Vector2.Zero;
14         velocity = Vector2.Zero;
15         Visible = true;
16     }
17
18     public virtual void HandleInput(InputHelper inputHelper) { }
19
20     public virtual void Update(GameTime gameTime)
21     {
22         Position += velocity * (float)gameTime.ElapsedGameTime.TotalSeconds;
23     }
24
25     public virtual void Draw(GameTime gameTime, SpriteBatch spriteBatch) { }
26
27     public virtual void Reset()
28     {
29         velocity = Vector2.Zero;
30     }
31 }
```

Listing 14.1 shows the full `GameObject` class. Go ahead and add this class to your project. You can type the code yourself or take the *GameObject.cs* file from the JewelJam3b project.

14.2.2 The *SpriteGameObject* Class

Next, let's create a subclass of `GameObject` that can represent a game object with a sprite. This `SpriteGameObject` class has two extra member variables: a `sprite` and an `origin`. So, the class should start out like this:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
```

```
class SpriteGameObject : GameObject
{
    protected Texture2D sprite;
    protected Vector2 origin;
    ...
}
```

Add a constructor method that takes one parameter: a string that indicates the name of the sprite to load. Inside the constructor, use the static property `ExtendedGame.ContentManager` (which you've created earlier) to load that sprite. Also assign a default value to the `origin` member variable.

```
public SpriteGameObject(string spriteName)
{
    sprite = ExtendedGame.ContentManager.Load<Texture2D>(spriteName);
    origin = Vector2.Zero;
}
```

Because a `SpriteGameObject` always has a `sprite`, you can also implement the `Draw` method. In this method, you draw the `sprite` at the object's current position and `origin`, but only if the object is currently visible:

```
public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    if (Visible)
    {
        spriteBatch.Draw(sprite, position, null, Color.White,
            0, origin, 1.Of, SpriteEffects.None, 0);
    }
}
```

Notice the keyword **override**: this is necessary, because we want the compiler to know that this `Draw` method is a *special version* of the `Draw` method defined by `GameObject`.

Listing 14.2 shows the full `SpriteGameObject` class. In our version, we've also added a `BoundingBox` property just like in the Painter game. This will be useful if we ever want to handle collisions between game objects. Finally, we've added two read-only properties, `Width` and `Height`, which return the width and height of this object's `sprite`. That way, other objects can ask for this object's size without having to access the `sprite` itself.

Listing 14.2 The SpriteGameObject class in JewelJam3b

```

1  using Microsoft.Xna.Framework;
2  using Microsoft.Xna.Framework.Graphics;
3
4  class SpriteGameObject : GameObject
5  {
6      protected Texture2D sprite;
7      protected Vector2 origin;
8
9      public SpriteGameObject(string spriteName)
10     {
11         sprite = ExtendedGame.ContentManager.Load<Texture2D>(spriteName);
12         origin = Vector2.Zero;
13     }
14
15     public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
16     {
17         if (Visible)
18         {
19             spriteBatch.Draw(sprite, Position, null, Color.White,
20                 0, origin, 1.0f, SpriteEffects.None, 0);
21         }
22     }
23
24     public int Width { get { return sprite.Width; } }
25
26     public int Height { get { return sprite.Height; } }
27
28     /// <summary>
29     /// Gets a Rectangle that describes this game object's current bounding box.
30     /// This is useful for collision detection.
31     /// </summary>
32     public Rectangle BoundingBox
33     {
34         get
35         {
36             // get the sprite's bounds
37             Rectangle spriteBounds = sprite.Bounds;
38             // add the object's position to it as an offset
39             spriteBounds.Offset(Position - origin);
40             return spriteBounds;
41         }
42     }
43 }

```

14.2.3 Describing the Game World as a List of Game Objects

You've now added two new classes that can represent many possible game objects. The final step is to represent the overall game world by a *list* of these objects.

Do you remember how the Painter game had separate member variables for each object? The Update method of the Painter class contained something like this:

```

cannon.Update(gameTime);
ball.Update(gameTime);
can1.Update(gameTime);

```

```
can2.Update(gameTime);
can3.Update(gameTime);
```

If we instead store the game world as a list of objects, in a single member variable, then this code will become much easier. We can then just call `Update` for all game objects in the list. We don't have to know how many objects there are or which object is which. All we care about is that all game objects will do something.

Thanks to the `GameObject` class that you've just created, you can now store all objects in a *list of game objects*—or more specifically, a variable of type `List<GameObject>`. You've already promised that each `GameObject` knows how to update and draw itself. The details of this updating and drawing may be different for each subclass of `GameObject`, but that doesn't matter. The compiler only needs to know that each object is *at least* a `GameObject`, but more specific classes are also allowed.

Start by giving the `ExtendedGame` class an extra member variable:

```
protected List<GameObject> gameWorld;
```

Initialize it in the `LoadContent` method:

```
gameWorld = new List<GameObject>();
```

With this list in place, the `Update`, `HandleInput`, and `Draw` methods become very easy to fill in. In each of these methods, you can use a **foreach** loop (which we've explained in the previous chapter) to let each object in the list do something.

Add the following code to the end of the `Update` method:

```
foreach (GameObject obj in gameWorld)
    obj.Update(gameTime);
```

This will make sure that all objects update themselves. Similarly, add the following code to the end of the `HandleInput` method:

```
foreach (GameObject obj in gameWorld)
    obj.HandleInput(inputHelper);
```

Notice that you're passing the game's `inputHelper` member variable as an argument to the `HandleInput` method of each object.

Finally, you can now give the `ExtendedGame` class its own `Draw` method:

```
protected override void Draw(GameTime gameTime)
{
    GraphicsDevice.Clear(Color.Black);

    // start drawing sprites, applying the scaling matrix
    spriteBatch.Begin(SpriteSortMode.Deferred, null, null, null, null, null, spriteBatch);

    // let all game objects draw themselves
    foreach (GameObject obj in gameWorld)
        obj.Draw(gameTime, spriteBatch);

    spriteBatch.End();
}
```

Next to the standard tasks such as starting and ending the sprite batch, this `Draw` method simply says that all objects in the game world should draw themselves. Eventually, this method will replace the `Draw` method of `JewelJam`. But before we can allow that, we'll have to make sure that all game objects in `Jewel Jam` are actually instances of `GameObject`.

14.2.4 Collections and Polymorphism

How can the methods you’ve just written work correctly? If each object is only referred to as a “simple” `GameObject`, and not as something more specific, how does the program know that each object may have its own specific version of `Update` or `Draw`?

Well, this is inheritance and **polymorphism** at its finest. In Chap. 10, we’ve explained that the Painter game allowed the following code:

```
ThreeColorGameObject obj = new Cannon(...);
```

and that the instruction `obj.Draw(...)` would then call the most specific version of the `Draw` method (in that case, `Cannon`’s version). In other words, even though `obj` is a variable of type `ThreeColorGameObject`, the program will automatically check if `obj` actually refers to something more specific than that.

The exact same concept applies to our list of game objects. In the following code:

```
foreach (GameObject obj in gameWorld)
    obj.Update(gameTime);
```

the variable `obj` will refer to a certain game object, in each iteration of the **foreach** loop. But every time you call `obj.Update`, the program will automatically check what kind of object `obj` *really* refers to. This shows once again how powerful inheritance really is!

14.3 Changing the Grid of Jewels

The `JewelJam` class inherits from `ExtendedGame`, so it also has access to the (protected) member variable `gameWorld` that you’ve just added. Let’s try to change `JewelJam` so that it uses this list of game objects instead of storing everything in separate variables.

For starters, you can remove the `background` member variable and replace it with a `SpriteGameObject`. That way, the `SpriteGameObject` itself will be in charge of loading and drawing the sprite. In the `LoadContent` method (just after the call to **base**.`LoadContent`), add this object to the game world:

```
SpriteGameObject background = new SpriteGameObject("spr.background");
gameWorld.Add(background);
```

You can now also remove the `Texture2D background` member variable, because that functionality is now wrapped inside a `SpriteGameObject`.

For the `Draw` method, it’d be nice if we could reuse the `Draw` method of `ExtendedGame` somehow. Unfortunately, this is still a bit difficult right now, because the *grid of jewels* is not represented by game objects yet. Currently, we’re storing a grid of integers and then drawing the correct sprites based on these integers. If we want to use our new classes instead, this setup will have to change. First, we need to turn each jewel into a kind of `GameObject`. Second, we need to think of a clever way to store these objects in a grid formation. Let’s solve these problems one by one.

14.3.1 Turning a Jewel into a Game Object

To solve the first problem, add a new `Jewel` class to the game. An instance of `Jewel` will represent a single jewel in the grid. Because a jewel has a sprite and a position, it makes sense to turn `Jewel` into a subclass of `SpriteGameObject`:

```
class Jewel : SpriteGameObject
```

In fact, the only other thing this class currently needs is a *constructor*. Remember that the constructor of `SpriteGameObject` requires a parameter that indicates the sprite to load. You could create a `Jewel` constructor that also takes a **string** parameter and simply passes it on to the base class:

```
public Jewel(string sprite) : base(sprite) {}
```

But there's a slightly nicer way to do it. Recall that we're currently representing a jewel by the number 0, 1, or 2. Based on that number, the jewel gets associated to the sprite `spr_single_jewel1`, `spr_single_jewel2`, or `spr_single_jewel3`. Inspired by this, you can also give the `Jewel` constructor a *number* as a parameter (instead of a full sprite name), and then automatically convert that number to the correct sprite name. The following constructor does that:

```
public Jewel(int type)
    : base("spr_single_jewel" + (type + 1))
{
}
```

Do you see how the expression `"spr_single_jewel" + (type + 1)` converts an integer (number) to the correct sprite name? And do you understand why the brackets around `type+1` are necessary here?

Later in the game, the jewel's number (0, 1, or 2) will actually become useful for checking if the grid contains valid combinations of jewels. Therefore, it's a good idea to store this number in a member variable or property, so that you can use it later. We suggest to go for a public property **int** `Type` with a private **set** component. This will prevent other objects from changing the number of a jewel. Go ahead and add such a property now:

```
public int Type { get; private set; }
```

and set its value inside the constructor, based on the parameter value that the constructor has received:

```
Type = type;
```

14.3.2 Turning the Grid of Jewels into a Game Object

For the second problem (storing the jewels in a grid), the easiest solution right now is to add another class that will represent the entire grid of jewels. Add a new class `JewelGrid` that inherits from `GameObject`. Give it four member variables: a 2D array of `Jewel` objects plus three integers for storing the grid width, grid height, and cell size. This is what the beginning of the class looks like:

```
class JewelGrid : GameObject
{
    Jewel[,] grid;
    int gridWidth, gridHeight, cellSize;
    ...
}
```

Add a constructor method that has three **int** parameters (for the width, height, and cell size) and one `Vector2` parameter (for the overall position of the grid). Inside this constructor, start by copying these parameters into the correct member variables:

```
public JewelGrid(int width, int height, int cellSize, Vector2 offset)
{
    // copy the width, height, and cell size
    gridWidth = width;
```

```

gridHeight = height;
this.cellSize = cellSize;
Position = offset;
...

```

After that, initialize the grid variable. We'll do this in the Reset method. Make sure to call that method in the constructor:

```
Reset();
```

The Reset method itself starts out as follows:

```

public override void Reset()
{
    grid = new Jewel[gridWidth, gridHeight];
    ...
}

```

This creates a new 2D array that can store Jewel objects. But because Jewel is a class type, all elements in this array will initially have the value **null**.

To fill this array with jewels, you can use a nested **for** loop, just like your JewelJam class already did. But this time, you'll fill the grid with Jewel objects instead of just numbers:

```

for (int x = 0; x < gridWidth; x++)
{
    for (int y = 0; y < gridHeight; y++)
    {
        grid[x, y] = new Jewel(ExtendedGame.Random.Next(3));
    }
}

```

Note that we're using the Random property from ExtendedGame here to draw random numbers.

We're almost done now, but there's one more thing that we need to pay attention to. Previously, the Draw method of JewelJam drew each sprite at the correct position via a double **for** loop. In the new version, we simply want to draw each SpriteGameObject at the position that it currently has. This means that you have to *set* the position of each Jewel object that you create. To do that, add the following line to the inner **for** loop:

```
grid[x, y].Position = Position + new Vector2(x * cellSize, y * cellSize);
```

This is comparable to what we used to do in the Draw method of JewelJam: converting the *x*- and *y*-coordinates of a grid cell to a position in the game world. But this time, we store it once as the position of a jewel, instead of calculating it inside the Draw method. Remember: in this code, the expression Position refers to the position of the grid itself, which serves as an offset for all jewels.

Next, give the JewelGrid class its own version of the Draw method. This method should simply draw all jewels at their current positions. It turns out that a **foreach** loop also works for 2D arrays. This makes the code surprisingly simple:

```

public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    foreach (Jewel jewel in grid)
        jewel.Draw(gameTime, spriteBatch);
}

```

Note that we don't need the counters of a regular (nested) **for** loop anymore. Each jewel has already received the correct position in the Reset method. When the program reaches the Draw method, it doesn't have to calculate these positions anymore.

14.3.3 Moving Behavior to the *JewelGrid* Class

The main *JewelJam* class can now start using a *JewelGrid* instead of its own 2D array of integers. Remove the member variable `int[,] grid` from that class. As a replacement, add a *JewelGrid* to the game world inside the `LoadContent` method:

```
JewelGrid grid = new JewelGrid(GridWidth, GridHeight, CellSize, GridOffset);
gameWorld.Add(grid);
```

Make sure to do this *after* you add the background object, because the first object in the list will be drawn first.

In the constructor of *JewelJam*, you can now remove the code that fills the grid, because the new *JewelGrid* class now takes care of that by itself. You can also remove the code that stores the three jewel sprites in a *Texture2D* array. After all, each jewel will now figure for itself which sprite it needs.

If you've done everything in the order that we've suggested so far, then the *JewelJam* class still contains a `MoveRowsDown` method. This method currently contains errors, because *JewelJam* no longer stores its own grid. Instead, `MoveRowsDown` should now become a task that only the *JewelGrid* class can do. So, move this method to the *JewelGrid* class. You need to change a few things to make this method work properly:

- Instead of the properties `GridWidth` and `GridHeight`, you can now use the member variables `gridWidth` and `gridHeight`.
- Instead of filling the top row with numbers, you should now fill it with random jewels (just like in `Reset`).
- Whenever you store a jewel at a certain grid position, you also need to set that jewel's `Position` property. This is because the `Draw` method no longer draws objects at the right position based on their grid coordinates. Instead, each game object now stores its own position. So, whenever a jewel gets added to the grid (or moves to a different grid cell), you need to calculate and set the position yourself.

As an exercise, try to make these changes yourself. You can look in the *JewelJam3b* example project for the complete answer. In our version, we've given *JewelGrid* a helper method that calculates and returns a *Vector2* object based on grid coordinates:

```
Vector2 GetCellPosition(int x, int y)
{
    return Position + new Vector2(x * cellSize, y * cellSize);
}
```

Whenever you need to calculate a new position, you can simply call `GetCellPosition` instead of copying and pasting the full expression. This leads to less duplicate code and reduces the chance of copy-and-paste errors.

After you've made the correct changes to `MoveRowsDown`, you can give *JewelGrid* its own version of `HandleInput`:

```
public override void HandleInput(InputHelper inputHelper)
{
    // when the player presses the spacebar, move all jewels one row down
    if (inputHelper.KeyPressed(Keys.Space))
        MoveRowsDown();
}
```

And as a result, you can now remove the `Update` method from the `JewelJam` class. After all, this method doesn't have to do anything special anymore, compared to `ExtendedGame`'s version. Because the grid is now part of the game world, the `Update` and `HandleInput` methods of the grid (and its jewels) will be called automatically.

The final method of `JewelJam` that still has errors is the `Draw` method. Good news, though: you can remove that method now! Just like with `Update`, now that you've moved all relevant code to other classes, the `JewelJam` class doesn't have to do anything extra here anymore. The `ExtendedGame` class takes care of starting and ending the sprite batch, and all individual game objects take care of drawing themselves.



CHECKPOINT: JewelJam3b

Compile and run the program again. The game should still behave in the exact same way—but you're now using a much more general structure, which you can reuse in the other games you're going to make.

Whew, that was a pretty big overhaul! We hope you get the main idea, though: by creating general classes for the game and its objects, the `JewelGrid` and `Jewel` classes can truly focus on the tasks that are specific for Jewel Jam. As a result, these classes aren't cluttered with general tasks anymore.

14.4 A Hierarchy of Game Objects

We can take this idea of an object list one step further. The `JewelGrid` class is an example of a game object that *contains other game objects*. In more complicated games, you'll find many more examples of objects containing other objects. For instance, imagine a city-building game where a house can contain furniture. If the player moves the entire house to a different position, then all furniture inside that house should move as well. But what if a single piece of furniture consists of multiple objects again, such as a bookcase that contains objects on its shelves?

This example suggests that there can be a complete **game-object hierarchy**. Objects can contain other objects, and this pattern can repeat itself many levels deep. As a programmer, you don't always know in advance how often this pattern will be repeated. Luckily, there's a programming concept called *recursion* that's very helpful in scenarios like this one.

Going back to Jewel Jam, we're currently in a bit of trouble if we ever want to move the `JewelGrid` to a different position during the game. Try it out yourself by changing the grid's position *after* you've created the grid:

```
grid.Position = new Vector2(300,100);
```

If you run the game again, you'll see that nothing has changed. This is because all jewels inside the grid have already received a position based on the *old* position of the grid. By changing the position of the grid itself, you've only moved the "container" object somewhere else, but you haven't moved the objects *inside* it.

It would be nice if we could do something about that. In this section, you'll make a few more changes to your game engine, to prepare it for even more complicated and dynamic games. By doing so, you'll also learn about *recursion*.

14.4.1 Relations Between Game Objects

To indicate that objects can contain each other, we'll make a number of changes to the code. First of all, each `GameObject` will store a reference to the object in which it is contained. We'll call that containing object the **parent object**. To store this reference, add a new property to the `GameObject` class:

```
public GameObject Parent { get; set; }
```

This reference is sometimes called a **parent pointer**. For objects that don't have a parent, this reference will be **null**. Remember that **null** is the default value of every variable that has a class type. So, if we don't do anything special, all game objects will have a parent pointer that stores **null**. This is OK for objects that don't *have* a parent object. For objects that *should* have a parent object, we can use this new property to set it.

In Jewel Jam, the Jewel objects in the grid should have the JewelGrid object as their parent, because the grid *contains* these objects. So, in the JewelGrid class, you should set the parent of each new Jewel instance that you create. This happens in two places, namely, in the double for-loop of the Reset method:

```
grid[x, y].Parent = this;
```

and in the MoveRowsDown method where you fill the top row with new jewels:

```
grid[x, 0].Parent = this;
```

This is another nice example of the keyword **this**. Recall that **this** refers to the instance that is currently executing a method. In this case, it refers to the single JewelGrid instance. So, by setting **this** as the parent pointer of each jewel, the jewels will correctly store the grid as their parent.

Also, note that we're only creating *references* to objects. We're not creating a full copy of the grid; all jewels will point to the same instance of JewelGrid.

The opposite of a parent object is called a **child object**. In the case of Jewel Jam, all Jewel instances are child objects (or "children," for short) of the JewelGrid instance. If an object "C" has another object "P" as its parent, then "C" is a child of "P," and vice versa.

Watch Out: "Parent Class" Versus "Parent Object" — In Chap. 10, we've used the term "parent class" as a different word for "superclass" or "base class." This term has to do with inheritance: it describes how classes can be *specific versions* of each other.

In *this* chapter, we're using the term "parent object" to denote the object in which another object is contained. This is related to the game-object hierarchy: it describes how objects can *contain* each other.

Although both concepts involve the term "parent" (and its opposite term "child"), they are two very different things. The hierarchy of game objects has nothing to do with inheritance!

The best way to avoid this confusion would be to stop using the term "parent class" completely, and just use "superclass" or "base class" instead. That way, the term "parent" will always refer to the game-object hierarchy. We'll try to do that in this book from now on. But of course, we can't control what *other* people write—so if you ever come across the word "parent" or "child" in a text about programming, pay close attention to what is really meant there.

14.4.2 Global and Local Positions

Now that our objects are connected in a hierarchy, our next task is to make sure that child objects (such as the jewels) will move along with their parent (such as the jewel grid) if that parent ever changes its position. Currently, each `GameObject` simply stores its *absolute* position in the game world, via the `Position` property. This is also called a **global position**. We'll change that so that each `GameObject` will store its *relative* position *inside* its parent object. This is also called a **local position**.

For example, the top-left jewel of the grid currently stores its global position, which is equal to the grid offset. Instead, it should store the *local* position (0, 0). This will indicate that the jewel lies in the top-left corner of the grid, no matter where the grid itself is located.

Whenever we draw a game object on the screen, we will *recalculate* the global position of that object. You can do that by taking the local position and adding it to the position of the parent object. And if that parent object has another parent object, we add that parent's position as well, and so on. So, to convert a local position to a global position, we should go all the way up in the game-object hierarchy.

In the `GameObject` class, rename the `Position` property to `LocalPosition`, to make extra clear what this property means. Then, in the `JewelGrid` class, you should now set the `LocalPosition` of each jewel (instead of the `Position`, which no longer exists). Also, you should make sure that the grid offset is no longer added to the position of each jewel. If you have the same helper method `GetCellPosition` that we've created, you can change that method as follows:

```
Vector2 GetCellPosition(int x, int y)
{
    return new Vector2(x * cellSize, y * cellSize);
}
```

In other words, simply remove the “Position +” part from your code.

Now that each object only stores its *local* position (relative to the parent object), the idea is to calculate the *global* position whenever you draw the object. How exactly should you calculate the global position of an object? If you think about it, there are two cases:

1. If an object does *not* have a parent, then its global position is simply equal to its local position. After all, there is no “containing object” to pay attention to.
2. If an object *does* have a parent, then its global position is equal to its local position *plus the global position of its parent*.

In other words, the global position of an object should always be “offset” by the global position of the parent object, unless that parent object doesn't exist.

You can express this in the `GameObject` class via a read-only property:

```
public Vector2 GlobalPosition
{
    get
    {
        if (Parent == null)
            return LocalPosition;
        return LocalPosition + Parent.GlobalPosition;
    }
}
```

Do you see what this code does? The `if` block corresponds to case 1, in which the object does not have a parent. The line after that corresponds to case 2, in which the object does have a parent. And

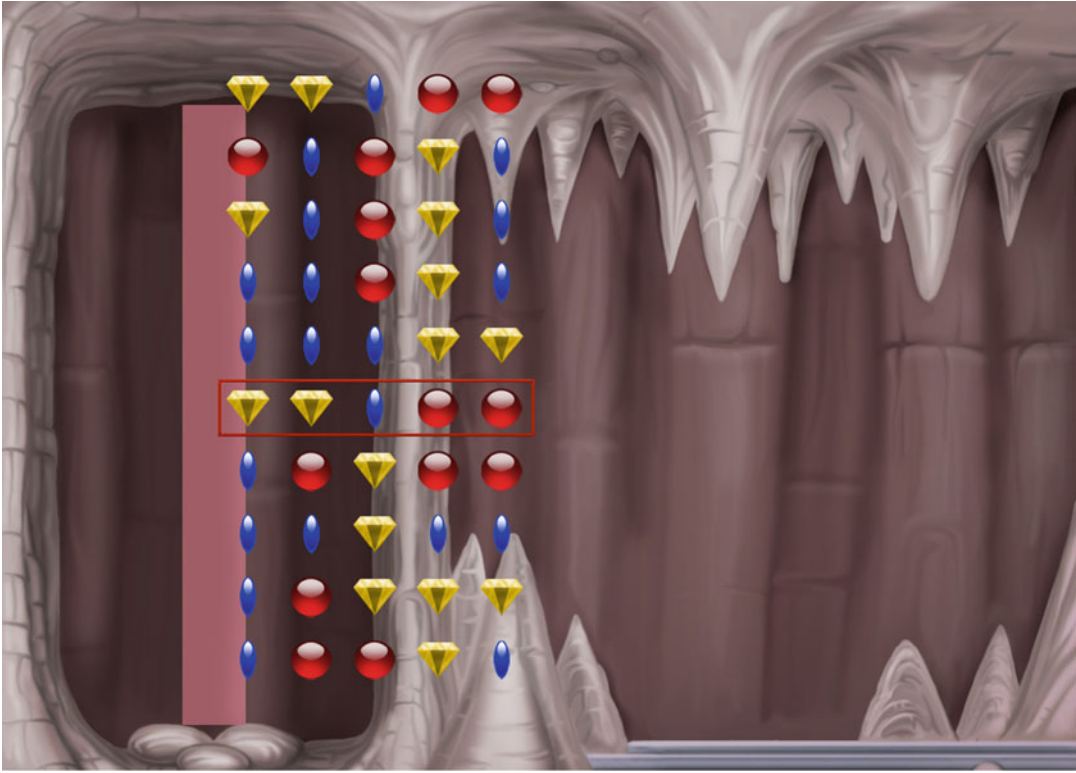


Fig. 14.1 Moving the grid to another position

that line for case 2 contains the expression `parent.GlobalPosition`. For example, whenever you ask for the `GlobalPosition` of a jewel, that jewel will ask for the `GlobalPosition` of the grid, and then add its own position to that result.

After this change, you should now draw all objects at their *global* position instead of their local position. Go to the `Draw` method of `SpriteGameObject`, and make sure that `GlobalPosition` is used for drawing the sprite:

```
spriteBatch.Draw(sprite, GlobalPosition, null, Color.White,
    0, origin, 1.0f, SpriteEffects.None, 0);
```

Similarly, change the `BoundingBox` property of `SpriteGameObject` so that it uses the global position.

Compile and run the program again. This time, if you change the position of the grid later on, you'll see that all jewels will shift along with it. That's because `GlobalPosition` will be recalculated in each frame, based on the most recent position of the parent object. Figure 14.1 shows what happens if you set the grid's `LocalPosition` to (300, 100).

Just for fun, try to see what happens if you give the grid a *velocity*! In the constructor of `JewelGrid`, set the velocity to something like `new Vector2(100,100)`. Run the game again and you'll see that the grid moves through the screen—and all jewels inside the grid move along with it. This shows the true power of placing game objects in a hierarchy.

14.4.3 Recursion: A Method or Property Calling Itself

The code of the `GlobalPosition` property may look strange to you: the property calls itself! This is perfectly valid C#, and it's actually a very powerful programming concept.

Imagine a game with a deeper hierarchy, such as our previously used example of a book on a shelf, in a bookcase, in a house, in a city. When we ask for the `GlobalPosition` of the book, this property will ask for the `GlobalPosition` of the shelf, which will then ask for the `GlobalPosition` of the bookcase, and so on. Thus, a call to the `GlobalPosition` property will *call that same property* multiple times, namely, once for each “layer” of the object hierarchy.

This concept is called **recursion**. (You can also say that the `GlobalPosition` property is *recursive*.) A property (or method) is recursive if it includes a call to that same property (or method). `GlobalPosition` is recursive because it uses *another* call to `GlobalPosition`. And this other call may trigger yet another call to `GlobalPosition` and so on—until we've reached the point where no recursion is possible anymore (in this case, when we've reached an object that doesn't have a parent).

Recursion can be pretty mind-boggling if you hear about it for the first time. Later in this chapter, we'll discuss this topic some more. For now, let's go back to finishing the game-object hierarchy.

14.4.4 The `GameObjectList` Class

The `JewelGrid` is our first example of an object that has multiple child objects. Because this will happen more often, it's good to create a special class for describing a collection of game objects. Each object in the collection will have that collection as its parent object. Also, the collection will have a position by which all elements can be offset.

To model this, create a new class `GameObjectList` that is a subclass of `GameObject`. That's right: a `GameObjectList` is also a `GameObject` itself! This allows us to set a `GameObjectList` as the parent of other objects. A regular `List<GameObject>` does not allow this, because such a list is *not* a `GameObject` itself.

Try to create this class yourself, and then look at the `JewelJam3c` example project for the solution. Here are some guidelines to help you get started. The `GameObjectList` class should have a single member variable:

```
List<GameObject> children;
```

Furthermore, the class should have the following methods:

- A constructor that initializes the `children` variable.
- An `AddChild` method that adds an object to `children` and assigns **this** as the parent of that object.
- A `HandleInput` method that uses a **foreach** loop to call `HandleInput` of all children.
- Similar methods for `Update`, `Draw`, and `Reset`.

Make sure to use the word **override** in the right places, so that the compiler knows that these methods are special versions of the methods that `GameObject` already defined. As an example, here is the `Update` method of `GameObjectList`:

```
public override void Update(GameTime gameTime)
{
    foreach (GameObject obj in children)
        obj.Update(gameTime);
}
```

In a way, this method is *recursive* as well. After all, if you call the `Update` method of a `GameObjectList` instance, this list will call the same `Update` method for all elements in that list. But these elements can

be any kind of `GameObject`, including a `GameObjectList` again. By making sure that `GameObjectList` is a subclass of `GameObject`, you can now create lists that contain lists that contain lists and so on. This is another nice example of recursion.

To see your new class in action, let's replace the game world by a `GameObjectList`. Go to the `ExtendedGame` class and replace this member variable:

```
protected List<GameObject> gameWorld;
```

by this one:

```
protected GameObjectList gameWorld;
```

The `HandleInput`, `Update`, and `Draw` methods of `ExtendedGame` currently loop over the list of game objects to call the corresponding method for each object. Instead, they should now simply pass that work onto the `gameWorld` variable. For example, the full `Update` method should now look like this:

```
protected override void Update(GameTime gameTime)
{
    HandleInput();
    gameWorld.Update(gameTime);
}
```

Finally, in the `JewelJam` class, you'll have to replace `gameWorld.Add` by `gameWorld.AddChild` a few times.

After these changes, you should be able to compile and run the game again, and everything should still work in the same way. But the big advantage is that objects are now nicely structured in a hierarchy with parent-child relationships.

To recap: there's now a single game world (a `GameObjectList`) that updates and draws itself. It will automatically make sure that all objects *inside* the game world update and draw themselves as well. And because these objects can be lists again, the entire game world can now be a complete hierarchy of game objects. You'll use this hierarchy much more in the later chapters of this book.

CHECKPOINT: JewelJam3c



In our example project, we've moved all classes of our "game engine" to a separate folder named *Engine*. These classes are `ExtendedGame`, `GameObject`, `GameObjectList`, `InputHelper`, and `SpriteGameObject`. This way, the code specific to Jewel Jam is nicely detached from the code that can be reused in other games. We encourage you to do the same.

This concludes the programming work for this chapter.

14.5 More on Recursion

To finish the chapter, let's take a closer look at what **recursion** really is. You've seen two examples of recursion so far:

- The `GlobalPosition` property of a game object calls the `GlobalPosition` property of the parent object (if that parent exists).
- The `Update` method of `GameObjectList` calls the `Update` method of all game objects inside that list. (The same goes for `HandleInput`, `Draw`, and `Reset`.)

In a way, a recursive method is similar to a loop: it “repeats” itself over and over again, up to a point where the recursion stops. It turns out that you can apply recursion to a lot of problems that you can also solve with *loops*. It’s really a matter of taste which style you prefer (loops or recursion). But sometimes, recursion makes a bit more sense than loops, or vice versa.

In this section, you’ll learn more about recursion. This section is rather advanced and more “mathematical” than the rest of this book. If you simply want to continue working on the Jewel Jam game, you can also go over this section more quickly and then move to the next chapter.

The exercises at the end of this chapter will let you train your recursion skills. Some of them are quite challenging!

14.5.1 The Two Ingredients of Recursion

A recursive method always contains two main parts:

1. A piece of code that calls the same method again but (for example) with different parameters. This part of the method is also called the **recursive case**.
2. A piece of code in which there is no recursion anymore, because you have “reached the end” in some sense. This part of the method is also called the **base case**, or the “stopping condition.” At the base case, it’s immediately clear what needs to happen, and you don’t need a recursive call to the same method anymore.

If this confuses you, think about the two examples you’ve seen so far.

In the `GlobalPosition` property, the base case occurs when you reach an object that doesn’t have a parent. In that case, `GlobalPosition` is simply equal to the local position of that object. If the object *does* have a parent, then its `GlobalPosition` depends on the `GlobalPosition` of the parent object.

In the `Update` method, the base case occurs when you reach an object that doesn’t have any child objects. (This happens when an object is an empty list or when it is not a `GameObjectList` at all.) In that case, you simply update the object itself, and that’s that. If the object *does* have children, then you use a **foreach** loop to let each child update itself.

14.5.2 Example: The Sum from 1 to n

To give you a better understanding of recursion, we’ll show you another example, unrelated to the Jewel Jam game. Take a look at the following method. Given an integer `n` that is larger than zero, this method uses a “good old” **for** loop the sum $1 + 2 + 3 + \dots + n$ and then returns that result.

```
int Sum(int n)
{
    int result = 0;
    for (int i=1; i<=n; i++)
        result += i;
    return result;
}
```

It’s possible to write a version of the `Sum` method that uses *recursion* instead of a loop. To do that, it’s useful to first give a *recursive definition* of what we want to calculate.

We already know that the parameter n is at least 1. If n is *exactly* 1, then the result of this method should simply be 1. If n is larger than one, take a look at what happens when you write out the entire formula. Here's an example for the number 5:

$$\text{Sum}(5) = 1 + 2 + 3 + 4 + 5$$

And here's an example for the number 4:

$$\text{Sum}(4) = 1 + 2 + 3 + 4$$

That's interesting: the only difference between these two examples is the "+5" part! This means that you could calculate $\text{Sum}(5)$ by *first* calculating $\text{Sum}(4)$ and then adding 5 to that result. And in turn, you could calculate $\text{Sum}(4)$ by first calculating $\text{Sum}(3)$ and then adding 4 and so on. At some point, you'll reach $\text{Sum}(1)$, for which the result is immediately clear (namely, 1).

This leads to the following recursive definition of Sum:

- If $n = 1$, then $\text{Sum}(n) = 1$.
- If $n > 1$, then $\text{Sum}(n) = \text{Sum}(n-1) + n$.

The first item of this definition is the *base case* where there is no more recursion. The second item is the *recursive case*: it uses a recursive call to Sum but with a smaller number as input.

Once you have a recursive definition for something (with a base case and a recursive case), it's pretty easy to translate that definition to (C#) code. Here's the recursive version of the Sum method:

```
int SumRecursive(int n)
{
    if (n == 1)
        return 1;
    return SumRecursive(n-1) + n;
}
```

Do you see how the first lines match the base case, and how the last line matches the recursive case? (Notice that we don't need the keyword **else** anymore, because the **if** block already ends with a **return** instruction.)

If you call $\text{SumRecursive}(5)$ somewhere in your code, then the program will find out that this is the same as $\text{SumRecursive}(4) + 5$, which is the same as $(\text{SumRecursive}(3) + 4) + 5$, and so on, until you reach $\text{SumRecursive}(1)$ which doesn't trigger a recursive call anymore.

Let's summarize what you've learned about recursion:



Quick Reference: Recursion

A recursive method is a method that calls itself. Such a method always contains two different cases:

- A *recursive case* in which the method calls itself but with different parameters (or a different calling object).
- A *base case* in which the method does not call itself anymore, because the parameters (or the object that receives the method call) are simple enough to calculate a result immediately.

You could see recursion as a different way of achieving a "looping" effect. If you can solve a problem with a **for** or **while** loop, you can often also solve it with recursion.

14.5.3 Watch Out for Infinite Recursion

The base case of a recursive method is very important. Without it, the method will never know when to stop, so it will keep calling itself forever. For example, if the `SumRecursive` method did not include a base case, then the program would think that `SumRecursive(1)` is equal to `SumRecursive(0) + 1`, and that `SumRecursive(0)` is equal to `SumRecursive(-1) + 0`, and that `SumRecursive(-1)` is equal to—you get the idea.

For the same reason, it's important that the recursive case always makes the problem a bit “easier,” so that you will eventually reach the base case. In the `SumRecursive` method, the expression `n-1` is crucial: it makes sure that the value of `n` will be smaller and smaller in each method call, until you reach the base case of 1.

If a recursive method never stops, then this can have two possible causes: either your base case is incorrect (or missing) or your recursive case doesn't make the problem easier. This effect is called **infinite recursion**, and it is very similar to an infinite loop (such as a **while** loop in which the loop condition never becomes **false**). Just like an infinite loop, infinite recursion will make your game crash.

Call Stack/Stack Overflow — While your game is running, the program keeps track of a so-called **call stack**: the sequence of method calls that haven't finished yet. A method gets added to the stack when you call it, and it gets removed from the stack when it has finished executing.

If your program executes a recursive method, then the call stack will contain that same method multiple times, stacked on top of each other. After all, the method repeatedly calls itself, and a method call cannot finish until its “sub-call” has finished first. But if (by accident) your recursive method never stops, the call stack becomes infinitely big.

If that happens, the program encounters a so-called **stack overflow**. This will trigger an error (or actually, an “exception”) that causes your program to crash.

That's the main difference between infinite *recursion* and an infinite *loop*. Infinite recursion automatically triggers an error, but an infinite loop does not. A loop doesn't contain any method calls that stack up over time, and each iteration of a loop is a “closed” piece of code that has to end before the next iteration can start. Because of that, an infinite loop cannot cause a stack overflow.

14.6 What You Have Learned

In this chapter, you have learned:

- how to use inheritance and collections to describe the game world as a list of game objects;
- how to extend this to a hierarchy of game objects, with global and local positions;
- how to use the concept of recursion (a method or property calling itself).

14.7 Exercises

1. *Hierarchies and Hierarchies*

What is the difference between a *class* hierarchy and a *game-object* hierarchy? For both concepts, give an example that we haven't already given in this book. Why is it important to *not* confuse one concept with the other?

2. *Global Positions Without Recursion*

The `GlobalPosition` property currently uses recursion to go all the way to the top of the game-object hierarchy. You could implement the same property *without* using recursion by replacing the recursive part by a **while** loop.

Write this non-recursive version of the `GlobalPosition` property.

3. *Loops and Recursion*

In the exercises of Chap. 9, you wrote a method `int Factorial(int n)` that calculates the *factorial* of a number *n*. As a reminder, the factorial 5 (often written with an exclamation mark as 5!) is defined as $5 \times 4 \times 3 \times 2 \times 1$, which is 120.

In this question, you'll write the same method in a different way. Again, you may assume that the value of *n* is at least 1.

- a. Write the `Factorial` method using a **for** loop from 1 to *n*. (Note: This is the same question as in Chap. 9, but it is worth revisiting now).
- b. Write the `Factorial` method using *recursion*.

Hint: Just like with the `Sum` example, first give a recursive definition of the factorial operator.

4. ** Recursive Reversal*

Write a recursive method `string Reverse(string s)` that returns the reversed version of a given string. For example, the method call `Reverse("Hello World!")` should return the string `!dlroW olleH`.

- a. For a first version, you can use the `Substring` method from the `String` class.
- b. If you're up for a challenge, try writing a version that doesn't use substrings. *Hint:* You can give your method extra **int** parameters that represent the current position in the input string. The recursive calls will only change those **int** parameters and not the **string** parameter.

5. ** Recursive Searching*

For this challenging exercise, it's useful to first do the exercise "Sorting and searching" from Chap. 13, which lets you write a method `int IndexOf(double[] a, double val)` that finds the first position of a value *val* in an array.

Now write a *recursive* version of that method. *Hint:* Give the method extra **int** parameters that indicate the part of the array in which you're looking.

Chapter 15

Gameplay Programming



In this chapter, you'll implement the gameplay of Jewel Jam, using the hierarchy of game objects that you've set up in the previous chapter. Most of this work will revolve around grid operations (using nested **for** loops) and communication between game objects. You will also give the game different *game states*, for example, to show a title screen and a help window. At the end of this chapter, the game will be completely playable.

You'll divide the work over multiple game objects. For example, there will be a separate RowSelector object for selecting a row in the grid, and a ScoreGameObject for displaying the score. This means that game objects will have to *communicate*: the RowSelector class needs to know about the JewelGrid class to ask for the grid's size, and the ScoreGameObject class needs to know the player's current score (which we'll store in the game world).

As you've seen in previous chapters, you can implement this game-object communication in many ways. To let a game object A know about another game object B, you could do any of the following things:

1. Add B as a parameter to a *method* of A, so that A knows about B during that method only.
2. Give A a permanent reference to B, for example, by passing B as a parameter in the *constructor* of A. This makes more sense if A needs B in multiple methods or if you cannot add other method parameters (due to inheritance) so option 1 does not work.
3. Let B do the actual work instead, and give A *properties* that B can use to change the data in A. This makes sense if the changes in A are very small.
4. If B is the game world, you can give the game a *static property* to retrieve the game world (as we did before in the Painter game). You can give the game world (read-only) *properties* for basic data, such as the score.
5. Create a mechanism so that all game objects can find each other. For instance, you could give each object a unique name, and then give the game world a method that searches for the object with a particular name.

In the first edition of this book, we implemented the last option. In this second edition, we'll avoid this "find object by name" mechanism. One reason to avoid it is that the mechanism could cause performance issues on the long term: each call to `GameWorld.Find(...)` would search through the entire game-object hierarchy. Another reason is that the unique names of objects were very sensitive to typos. But most importantly: if you choose any of the other solutions, you have to *think carefully* about which objects know what about each other. This will lead to nicer code that's easier to understand, and it will help you become a better programmer.

The solutions that we'll propose are definitely not the *only* possible solutions. To a large extent, which solution you pick is really a matter of taste, and each programmer will have different preferences. If you feel confident enough to try out *other* solutions than the ones we give, feel free to do so. However, the book will assume that you follow our examples, so keep that in mind if your code gets out of sync with ours.

15.1 Selecting Rows and Moving the Jewels

Let's start by implementing the object that selects a row in the grid. The player should be able to press the up and down arrow keys to select a different row, and they should be able to press Left and Right to shift the jewels in the selected row.

15.1.1 Class Overview and Constructor

Add a class `RowSelector` that extends the `SpriteGameObject` class. Also use the Pipeline Tool to add the sprite `spr_selector_frame.png` to your project. That's the sprite that `RowSelector` should use.

We'll give `RowSelector` two member variables. The first is an `int` that stores the row that is currently being selected, with an initial value of 0 (indicating the top row). The second is a reference to the `JewelGrid` of the game. We've made this choice because `RowSelector` clearly needs to know about the grid. However, many other game objects *don't* have to know about the grid, so it makes sense to keep this knowledge exclusive to `RowSelector` for now. So, in short, `RowSelector` has the following member variables:

```
int selectedRow;
JewelGrid grid;
```

To fill the grid member variable correctly, we'll pass the `JewelGrid` object as a parameter in the *constructor* of `RowSelector`. Thus, the header of the constructor looks like this:

```
public RowSelector(JewelGrid grid) : base("spr_selector_frame")
```

(Remember that the base class, `SpriteGameObject`, asks for a sprite name in its own constructor.) The body of the constructor should set the grid reference, the selected row index, and the sprite origin:

```
this.grid = grid;
selectedRow = 0;
origin = new Vector2(10,10);
```

We've chosen this sprite origin so that the sprite wraps nicely around the selected row. Also, we're using the keyword **this** here to resolve a name conflict. In the constructor, the name `grid` is used for two different things: a method parameter and a member variable. The expression `this.grid` refers to the member variable, whereas `grid` refers to the method parameter.

15.1.2 Adding a RowSelector to the Game World

To add an actual `RowSelector` instance to the game world, go to the `LoadContent` method of `JewelJam`. It would be nice if the `RowSelector` could use the same offset position as the grid itself. One way to

achieve this is to put the grid and the row selector inside a *parent* object. This parent object will be an instance of the `GameObjectList` class that you created in the previous chapter. If you give that parent object the position `GridOffset`, then both child objects (the grid and the row selector) can have a position of `(0, 0)` inside that parent.

To achieve this, replace the following code:

```
JewelGrid grid = new JewelGrid(GridWidth, GridHeight, CellSize, GridOffset);
gameWorld.AddChild(grid);
```

by this:

```
// add a "playing field" parent object for the grid and the row selector
GameObjectList playingField = new GameObjectList();
playingField.LocalPosition = GridOffset;
gameWorld.AddChild(playingField);

// add the grid to the playing field
JewelGrid grid = new JewelGrid(GridWidth, GridHeight, CellSize, Vector2.Zero);
playingField.AddChild(grid);

// add the row selector to the playing field
playingField.AddChild(new RowSelector(grid));
```

As you can see, we're now using `GridOffset` as the position of `playingField` (the parent object). In exchange, the grid itself now simply receives a position of `Vector2.Zero`. In fact, `Vector2.Zero` is already the default position of every `GameObject`, so you might as well remove the fourth parameter from the `JewelGrid` constructor. We've done this in the `JewelJam4a` example code.

If you compile and run the game now, you should see the row selector at the correct position, wrapped nicely around the top row of the grid. Next up, we should allow the player to *control* the row selector with the arrow keys.

15.1.3 Changing the Selected Row

Let's fill in the `HandleInput` method of `RowSelector` class:

```
public override void HandleInput(InputHelper inputHelper)
```

First of all, pressing the up or down arrow key should cause the selected row index to decrease or increase. So, the body of `HandleInput` could start as follows:

```
if (inputHelper.KeyPressed(Keys.Up))
    selectedRow--;
else if (inputHelper.KeyPressed(Keys.Down))
    selectedRow++;
```

You need to make sure that `selectedRow` always refers to a row that really exists. For example, row -1 does not exist (because 0 is the top row), and row 100 clearly doesn't exist either (because the grid only has ten rows). You can use a simple `if` instruction to make sure that `selectedRow` is at least 0:

```
if (selectedRow < 0)
    selectedRow = 0;
```

For the *maximum* value of `selectedRow`, the `RowSelector` should know about the height of the grid. Luckily, this class already have a grid member variable that points to the `JewelGrid` instance. We just have to give the `JewelGrid` class a property so that the `RowSelector` can ask for its height.

In `JewelGrid`, we suggest to replace the member variable `gridHeight` by a property:

```
public int Height { get; private set; }
```

While you're at it, do the same thing for the grid's *width*. You'll have to change the `JewelGrid` class a bit to reflect those changes, but that shouldn't be too difficult.

With these properties in place, the `RowSelector` class can write `grid.Height` to retrieve the height of the grid. Remember that arrays start counting at 0, so `grid.Height - 1` is the largest valid value of `selectedRow`. This leads to the following extra `if` instruction:

```
if (selectedRow > grid.Height - 1)
    selectedRow = grid.Height - 1;
```

In fact, `MonoGame` has a nice helper method that makes sure a number does not exceed a minimum and a maximum value. Cutting off a number like this is also called **clamping**. `MonoGame`'s helper method allows you to write the current two `if` statements with just a single line of code.

The helper method is called `MathHelper.Clamp`, and it takes three arguments: the number you want to clamp, the lowest allowed value, and the highest allowed value. So, you can replace your two `if` instructions by the following instruction:

```
selectedRow = MathHelper.Clamp(selectedRow, 0, grid.Height - 1);
```

This correctly updates the `selectedRow` variable. After that, the `RowSelector` object should update its position in the game world, so that it aligns nicely with the selected row. Remember that `RowSelector` starts out at position (0, 0) to match the top row. If the selected row changes, only the *y*-coordinate of the object needs to change, and the *x*-coordinate can stay 0.

How should you convert `selectedRow` to the correct *y*-coordinate in the game world? Well, you may remember that `JewelGrid` has a nice helper method, `GetCellPosition`, which converts grid coordinates to world coordinates. This is a good opportunity to make that method *publicly* accessible:

```
public Vector2 GetCellPosition(int x, int y)
{
    return new Vector2(x * cellSize, y * cellSize);
}
```

You can then use that method to align the `RowSelector` with grid position (0, `selectedRow`):

```
LocalPosition = grid.GetCellPosition(0, selectedRow);
```

Compile and run the game again. You should now be able to move the row selector up and down, without ever moving it outside the grid.

15.1.4 Shifting the Jewels in a Row

The final thing that the `HandleInput` method should do is *shift* the jewels in a row when the player presses the left or right arrow keys. In our opinion, it's better if the *grid itself* is in charge of shifting the jewels. The `RowSelector` should only *tell* the grid to do something when an arrow key is pressed, without worrying about the details. To achieve that, add the following instructions to `HandleInput`:

```

if (inputHelper.KeyPressed(Keys.Left))
    grid.ShiftRowLeft(selectedRow);
else if (inputHelper.KeyPressed(Keys.Right))
    grid.ShiftRowRight(selectedRow);

```

Of course, `ShiftRowLeft` and `ShiftRowRight` are methods that you'll have to add to the `JewelGrid` class. We'll show you how to write `ShiftRowLeft`; try to write the other method yourself as an exercise. `ShiftRowLeft` has the following header:

```
public void ShiftRowLeft(int selectedRow)
```

What should the body of this method look like? Well, the concept is actually very similar to the `MoveRowsDown` method that we've created before. To shift all jewels (on this row) to the left, you need to replace all jewels by their right neighbor. You have to do this in the correct order, to prevent all jewels from accidentally becoming the same. So far, the story is similar to `MoveRowsDown`, but this time, the movement is horizontal instead of vertical, and we only do it on a single row.

The main extra feature is that `ShiftRowLeft` should have a "rotational" effect: the jewel that used to be on the far *left* should reappear on the far *right*. To enable that, you should store a backup of the leftmost jewel *before* you start copying the jewels around.

Here is the full `ShiftRowLeft` method:

```

public void ShiftRowLeft(int selectedRow)
{
    // store the leftmost jewel as a backup
    Jewel first = grid[0, selectedRow];

    // replace all jewels by their right neighbor
    for (int x = 0; x < Width - 1; x++)
    {
        grid[x, selectedRow] = grid[x + 1, selectedRow];
        grid[x, selectedRow].LocalPosition = GetCellPosition(x, selectedRow);
    }

    // re-insert the old leftmost jewel on the right
    grid[Width - 1, selectedRow] = first;
    grid[Width - 1, selectedRow].LocalPosition = GetCellPosition(Width - 1, selectedRow);
}

```

Take your time to understand how this method works. Notice how we're also updating the `LocalPosition` of each jewel that moves to a different cell. This is required because the `grid` member variable only assigns *grid* coordinates to the jewels and not *world* coordinates. As a programmer, you're in charge of keeping these things in sync.

Try to write the `ShiftRowRight` method yourself. It's basically a mirrored version of `ShiftRowLeft`. This method should first store the *rightmost* jewel as a backup, then replace all jewels by their *left* neighbor, and finally reinsert the backup jewel on the *left* side. Take a look at the `JewelJam4a` example project if you get stuck.



CHECKPOINT: JewelJam4a

Once you've written both methods, compile and run the game again. By pressing the left and right arrow keys, you can now shift the jewels inside the selected row.

15.2 More Types of Jewels

The other thing the player can (eventually) do is press the spacebar to remove valid combinations of jewels. But so far, we've used a simplified version of a jewel that can only have three different values. In the full version of the game, there are 27 different jewels: all possible combinations of three colors, shapes, and numbers. It's time to change the `Jewel` class so that it supports all possible jewel types.

15.2.1 One Sprite Sheet for All Jewels

Remove the three existing jewel sprites from your game, and add the sprite `spr-jewels.png` instead. This image is also shown in Fig. 15.1. It contains all possible jewel sprites in a single image. Such an image is also called a **sprite sheet**. We'll talk more about sprite sheets in the next part of this book.

It turns out that there's another version of `spriteBatch.Draw` that allows you to draw only a certain *part* of a sprite. You can use this to your advantage: all instances of `Jewel` will use the same sprite, but they will draw a different part of it, based on their type. Let's work towards that now.

15.2.2 Three Properties of a Jewel

Instead of a single number between 0 and 2, a `Jewel` instance should now store *three* numbers between 0 and 2: one for the *color*, another for the *shape*, and another for the *number* of jewels. So, replace the single `Type` property by the following three:

```
public int ColorType { get; private set; }
public int ShapeType { get; private set; }
public int NumberType { get; private set; }
```

Now that we have to change the `Jewel` constructor anyway, let's make it responsible for drawing its own random numbers. In other words, the `Jewel` constructor won't have any parameters anymore, and it will use `ExtendedGame.Random` itself to set the three member variables:

```
public Jewel() : base("spr-jewels")
{
    ColorType = ExtendedGame.Random.Next(3);
    ShapeType = ExtendedGame.Random.Next(3);
    NumberType = ExtendedGame.Random.Next(3);
}
```

Observe that we're now also simply loading the `spr-jewels` sprite all the time, instead of a different sprite per jewel.

By removing the constructor's parameter, you've caused errors in the `JewelGrid` class where you create instances of `Jewel`. Fix these errors by writing `new Jewel()` instead of `new Jewel(ExtendedGame.Random.Next(3))` where necessary.



Fig. 15.1 A single sprite that contains all possible jewels

15.2.3 Drawing the Correct Part of the Sprite Sheet

The other version of `spriteBatch.Draw` that we mentioned earlier has an extra parameter of type `Rectangle`. This rectangle represents the part of the sprite that you want to draw. To prevent you from having to recalculate this `Rectangle` in every frame, give the `Jewel` class an extra member variable:

```
Rectangle spriteRectangle;
```

You’ve used `MonoGame’s Rectangle` struct before, to represent the bounding box of a sprite. It has a constructor with four parameters: the left boundary, the top boundary, the width, and the height.

Let’s initialize this rectangle in the constructor of `Jewel`. Remember: we have a “sprite sheet” of 27 possible jewels, which is stored in the `sprite` member variable that `Jewel` inherits from `SpriteGameObject`. We want to select the correct square within that sprite sheet to use and store that square inside the `spriteRectangle` variable.

The top boundary of this square is always 0, because our square should always start at the top of the sprite. The height of this square is always `sprite.Height`, because we want to cover the full height of the sprite. The width of the square is *also* `sprite.Height`, because that’s the width of each individual jewel inside the sprite sheet.

The only thing that may vary is the *left boundary* of the square. For now, start by always filling in a value of 0, as follows:

```
spriteRectangle = new Rectangle(0, 0, sprite.Height, sprite.Height);
```

The `Jewel` class should now override the `Draw` method, because the “standard” `Draw` method of `SpriteGameObject` is no longer good enough. The method should look like this:

```
public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    // draw the correct sprite part at the jewel's position
    spriteBatch.Draw(sprite, GlobalPosition, spriteRectangle, Color.White);
}
```

This calls the special version of `spriteBatch.Draw` that we mentioned earlier. As you can see, the third parameter is the `Rectangle` that we’ve just calculated.

Compile and run the game now. You’ll see that all jewels look like a single yellow diamond. This is because `spriteRectangle` always has a left boundary of 0, so we’re always drawing the leftmost square of the sprite *spr-jewels.png*. The yellow diamond happens to be the leftmost jewel that appears in that sprite.

Now, how should we change the left boundary of `spriteRectangle` so that it depends on a jewel’s `ColorType`, `ShapeType`, and `NumberType`? Let’s convert these three numbers to a single *index* (between 0 and 26) that determines the exact jewel to draw. For example, the triple blue diamond lies at index 11, and the single red sphere lies at index 24. If you look carefully at the sprite sheet, you’ll see the following pattern:

- All jewels of a certain *color* are grouped in blocks of nine jewels. Let’s say we let the yellow jewels correspond to a `ColorType` of 0, the blue jewels to a `ColorType` of 1, and the red jewels to a `ColorType` of 2. This means that the index of a particular `Jewel` should be at least $9 * \text{ColorType}$.
- Within such a block of nine jewels with a certain color, all jewels of a certain *shape* are grouped in blocks of three. Let’s say the diamonds correspond to a `ShapeType` of 0, the ellipses to a `ShapeType` of 1, and the spheres to a `ShapeType` of 2. This means that the index of a particular `Jewel` should be offset by another $3 * \text{ShapeType}$.

- This narrows our search to a block of three jewels: all with the same color and shape, but with a different number. To choose the correct sprite from those three, the index should be offset further by `NumberType`.

In summary, the expression `9 * ColorType + 3 * ShapeType + NumberType` converts the three jewel properties to the correct index in the sprite sheet. What remains is to multiply this index by `sprite.Height`, because that's the width of each jewel image in the sprite sheet. Thus, the following code should go into your `Jewel` constructor:

```
int index = 9 * ColorType + 3 * ShapeType + NumberType;
spriteRectangle = new Rectangle(index * sprite.Height, 0, sprite.Height, sprite.Height);
```

Compile and run the program again. You should now see jewels of many different types. Nice work!

15.3 Finding Combinations of Jewels

Now that you've upgraded the `Jewel` class so that it has three properties, you can start discovering valid combinations of jewels in the grid. In the `JewelGrid` class, add a method with the following header:

```
bool IsValidCombination(Jewel a, Jewel b, Jewel c)
```

Given three `Jewel` instances, this method should return **true** if the jewels form a valid combination based on their color, shape, and number. If the jewels do not form a valid combination, the method should return **false**.

15.3.1 Checking if a Combination Is Valid

If you don't remember what a valid combination of jewels is, take a look at our explanation of the Jewel Jam game (in the introduction of Part III). A combination is valid if each property (color, shape, and number) is either the *same* for all jewels or *different* for all jewels. In our code, the three properties are all stored as integers. The following helper method checks if three integers are equal:

```
bool AllEqual(int a, int b, int c)
{
    return a == b && b == c;
}
```

And the following helper method checks if three integers are all different from each other:

```
bool AllDifferent(int a, int b, int c)
{
    return a != b && b != c && a != c;
}
```

You can group these into another helper method, which checks if three numbers are all equal *or* all different:

```
bool IsConditionValid(int a, int b, int c)
{
    return AllEqual(a, b, c) || AllDifferent(a, b, c);
}
```

Now, a combination of jewels is valid if the `IsValidCombination` method returns **true** for all three properties: the colors, the shapes, *and* the numbers. This means that the `IsValidCombination` method looks like this:

```
bool IsValidCombination(Jewel a, Jewel b, Jewel c)
{
    return IsValid(a.ColorType, b.ColorType, c.ColorType)
    && IsValid(a.ShapeType, b.ShapeType, c.ShapeType)
    && IsValid(a.NumberType, b.NumberType, c.NumberType);
}
```

Do you see how we're using the same helper methods for all three properties? This is possible because all properties are simply stored as an **int**. The helper methods don't need to know the *meaning* of these values; all they have to do is check for (in)equality.

15.3.2 Handling Keyboard Input

You've now finished the code that checks a combination of jewels. Next, you should call this method in some way when the player presses the spacebar. Because this has to do with keyboard input, it makes sense to write this code in the `HandleInput` method of `JewelGrid`. Currently, this method still calls `MoveRowsDown` when the spacebar is pressed, but that isn't really supposed to be part of the game. Let's clear the entire body of `HandleInput` and start from scratch:

```
public override void HandleInput(InputHelper inputHelper)
{
}
```

The method should only do something when the player presses the spacebar. You can check that in the beginning of the method, as follows:

```
if (!inputHelper.KeyPressed(Keys.Space))
    return;
```

Next, the method should scan the entire *middle column* of the grid for valid combinations of jewels. You can start by calculating the index of that column:

```
int mid = Width / 2;
```

Furthermore, we're only interested in combinations of *three jewels in a row*. This means that you can perform a single **for** loop, starting at row 0 and moving down from there. In each iteration of the loop, we want to check if the jewel on the current row forms a valid combination with the two jewels *below* it. These three jewels are given by the expressions `grid[mid, y]`, `grid[mid, y+1]`, and `grid[mid, y+2]`. Therefore, the loop should look like this:

```
for (int y = 0; y < Height - 2; y++)
{
    if (IsValidCombination(grid[mid, y], grid[mid, y+1], grid[mid, y+2]))
    {
        ...
    }
}
```

Notice the loop condition `y < gridHeight - 2`. This is because `gridHeight - 3` is the last row that still has two rows below it.

Now, *if* there is indeed a valid combination, the following things should happen:

- The three jewels should disappear.
- If there are any jewels above the trio that has disappeared, then those jewels should fall down.
- There's now a three-jewel gap at the top of the grid, which should be filled with three new random jewels.

For now, let's assume we have a `RemoveJewel` method that does exactly this for *one* jewel. That is, the method removes a single jewel, lets the jewels above it fall down, and then adds a new random jewel at the top of the grid. We'll create this method later. It will have two `int` parameters, indicating the grid coordinates of the jewel to remove.

Given this method, the `if` block of our loop becomes easy to fill in:

```
RemoveJewel(mid, y);
RemoveJewel(mid, y+1);
RemoveJewel(mid, y+2);
```

There's one catch: moving the jewels around (and adding new ones) may cause *new* combinations to appear. We want to ignore these new combinations, because these "chain reactions" can make the game unpredictable and difficult to test. This means that, after you've removed three jewels, you should skip two additional rows in the loop:

```
y += 2;
```

The `for` loop itself will perform another `y++`, so you'll end up skipping exactly the three rows that contained the trio you've just removed.

15.3.3 Removing Jewels from the Grid

Finally, implement the `RemoveJewel` method. This method contains (again) similar elements as `MoveRowsDown`. Given a certain grid position (x, y) , it should move all jewels above (x, y) one row down. Finally, it should fill the grid position $(x, 0)$ with a new random jewel.

See if you can work out the details of this method yourself. If you're unsure about your answer, take a look at the `JewelJam4b` example project.

This concludes the work we need to do in `JewelGrid`. Compile and run the game again. For the first time, you can now actually *play* the game as intended! But there's not much of a fun factor to the game yet: finding valid combinations doesn't earn you any points, and there's no timer that causes the game to end. We'll add those features next. Good news: they're much easier to add than the complicated grid logic from *this* section.



CHECKPOINT: JewelJam4b

In our example project, we've given `JewelGrid` an extra method `AddJewel` that combines all the instructions for placing a new jewel at a certain grid position.

By the way, you can now remove the old `MoveRowsDown` method, because we don't need it anymore.

15.4 Maintaining and Showing the Score

In this section, you'll extend the game so that it keeps track of the player's score. The player should earn 10 points whenever he/she finds a valid combination of jewels. Furthermore, the current score should be displayed on the screen, just like in the Painter game.

15.4.1 A Separate Class for the Game World

Soon, other game objects need to be able to get and set the player's score. To prepare for that, it's useful to create a separate class that will store everything related to the game world. This class, called `JewelJamGameWorld`, will basically do most of the work that the `JewelJam` class used to do.

`JewelJamGameWorld` is a subclass of `GameObjectList`, so it inherits the list of objects and all related methods. On top of that, we give it a property `Score` that stores the current score (which gets reset to 0 in the `Reset` method) and a method `AddScore` that adds a number of points to the score. The full class is given in Listing 15.1.

Listing 15.1 The `JewelJamGameWorld` class

```

1  using Microsoft.Xna.Framework;
2
3  class JewelJamGameWorld : GameObjectList
4  {
5      const int GridWidth = 5;
6      const int GridHeight = 10;
7      const int CellSize = 85;
8
9      // The size of the game world, in game units.
10     public Point Size { get; private set; }
11
12     // The player's current score.
13     public int Score { get; private set; }
14
15     public JewelJamGameWorld()
16     {
17         // add the background
18         SpriteGameObject background = new SpriteGameObject("spr_background");
19         Size = new Point(background.Width, background.Height);
20         AddChild(background);
21
22         // add a "playing field" parent object for the grid and all related objects
23         GameObjectList playingField = new GameObjectList();
24         playingField.LocalPosition = new Vector2(85, 150);
25         AddChild(playingField);
26
27         // add the grid to the playing field
28         JewelGrid grid = new JewelGrid(GridWidth, GridHeight, CellSize);
29         playingField.AddChild(grid);
30
31         // add the row selector to the playing field
32         playingField.AddChild(new RowSelector(grid));
33
34         // reset some game parameters
35         Reset();
36     }
37 
```

```

38 public void AddScore(int points)
39 {
40     Score += points;
41 }
42
43 public override void Reset()
44 {
45     base.Reset();
46     Score = 0;
47 }
48 }

```

With this new class in place, the `JewelJam` class itself can become much simpler. You can remove all member variables. In the `LoadContent` method, you can now create a new `JewelJamGameWorld` instance and store it in the `gameWorld` variable:

```
gameWorld = new JewelJamGameWorld();
```

This is allowed thanks to inheritance. After creating the game world, you still need to update the `worldSize` member variable afterwards. We could use the background sprite for that up until now, but that sprite is now hidden inside the `JewelJamGameWorld` object. Therefore, we should now ask the game world what size it has.

To make that possible, we've given `JewelJamGameWorld` a property `Size` that contains the world size (which is the size of the background sprite), as you can see in Listing 15.1 as well. We'd like to use this property in the `LoadContent` method of `JewelJam`, as follows:

```
worldSize = gameWorld.Size;
```

Unfortunately, the compiler won't allow that. Why is that? Well, as far as the `ExtendedGame` class is concerned, the `gameWorld` member variable simply stores a `GameObjectList`. In the case of `JewelJam`, we know that it's something more specific than that, namely, a `JewelJamGameWorld`, but the compiler doesn't see that automatically. We can only use the `gameWorld` variable for the things that `GameObjectList` knows about.

To work around this, the trick is to *cast* the `gameWorld` variable to a `JewelJamGameWorld` object here. When you do that, you *can* suddenly use the `Size` property:

```
worldSize = (JewelJamGameWorld)gameWorld.Size;
```

The complete `LoadContent` method should now look like this:

```

protected override void LoadContent()
{
    base.LoadContent();

    // initialize the game world
    gameWorld = new JewelJamGameWorld();

    // apply scaling with the new world size
    worldSize = (JewelJamGameWorld)gameWorld.Size;
    FullScreen = false;
}

```

But remember: our final goal of creating a `JewelJamGameWorld` class was to make sure that every game object can get a reference to the game world. We'll do that by using the keyword **static** in the right places, just like in the `Painter` game.

First, go to `ExtendedGame` and make the `gameWorld` member variable static:

```
protected static GameObjectList gameWorld;
```

Next, give the JewelJam class the following static property:

```
public static JewelJamGameWorld GameWorld
{
    get { return (JewelJamGameWorld)gameWorld; }
}
```

This way, all game objects can write JewelJam.GameWorld to retrieve the game world. Again, we're using a *cast* to explicitly convert the game world to a JewelJamGameWorld. For the same reason as before, this is required if we want to use game-world features that are specific for Jewel Jam. Without this cast, JewelJam.GameWorld would just return a GameObjectList, which (for example) does not have the Score property.

You can even use this same GameWorld property in the LoadContent method now, so that you only have to write the cast once.

15.4.2 Updating the Score

You're now finally ready to update the score from within other classes. In this case, the place where the player can score points is the HandleInput method of JewelGrid. Inside the **if** block that already removes a valid combination of jewels, add the following line:

```
JewelJam.GameWorld.AddScore(10);
```

This will add 10 points for each valid combination that gets removed.



CHECKPOINT: JewelJam4c

Compile and run the game again. You can now score points, although you can't see this yet as a player.

15.4.3 The TextGameObject Class

To display the score on the screen, you could draw the current score just like how you drew the score in the Painter game. However, to make our game engine even more reusable, let's go a step further.

Listing 15.2 shows the TextGameObject class. This is a subclass of GameObject, specifically meant for game objects that only consist of text (instead of a sprite). Feel free to copy this class directly into the Engine folder of your project. The class is also available in the JewelJam4d example project. Here's a quick rundown of what the class does:

Listing 15.2 The TextGameObject class

```
1 using Microsoft.Xna.Framework;
2 using Microsoft.Xna.Framework.Graphics;
3
4 class TextGameObject : GameObject
5 {
6     protected SpriteFont font;
7     protected Color color;
8     public string Text { get; set; }
```

```

9
10 public enum Alignment
11 {
12     Left, Right, Center
13 }
14
15 protected Alignment alignment;
16
17 public TextGameObject(string fontName, Color color,
18     Alignment alignment = Alignment.Left)
19 {
20     font = ExtendedGame.ContentManager.Load<SpriteFont>(fontName);
21     this.color = color;
22     this.alignment = alignment;
23
24     Text = "";
25 }
26
27
28 public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
29 {
30     if (!Visible)
31         return;
32
33     // calculate the origin
34     Vector2 origin = new Vector2(OriginX, 0);
35
36     // draw the text
37     spriteBatch.DrawString(font, Text, GlobalPosition,
38         color, 0f, origin, 1, SpriteEffects.None, 0);
39 }
40
41 float OriginX
42 {
43     get
44     {
45         if (alignment == Alignment.Left) // left-aligned
46             return 0;
47
48         if (alignment == Alignment.Right) // right-aligned
49             return font.MeasureString(Text).X;
50
51         return font.MeasureString(Text).X / 2.0f; // centered
52     }
53 }
54 }

```

- An instance of `TextGameObject` stores a font, a color, a text to draw, and an alignment. The text can be changed via a public property. The font, color, and alignment need to be given to the constructor.
- The alignment indicates if the text should be left-aligned, right-aligned, or centered. To represent this, we've added a simple **enum** with three possible values.
- The constructor of `TextGameObject` loads the font with a given name, and it initializes the member variables based on the other constructor parameters.
- The `Draw` method draws the text at the game object's position (if the object is visible). It also calculates an offset based on the desired alignment of the text.
- The alignment is calculated based on a (private) property `OriginX`. This property uses a nice helper method from `MonoGame`: `MeasureString` from the `SpriteFont` class. For a particular font, the

`MeasureString` method calculates how large a piece of text will be when shown on the screen. This returns a `Vector2` containing the width and height of the text. If the text should be left-aligned, we can ignore this size. If the text should be right-aligned, we need to use the text's width as the origin. If the text should be centered, we need an origin that lies exactly in the middle.

15.4.4 Showing the Score on the Screen

Next, create a class `ScoreGameObject` that inherits from `TextGameObject`. In its constructor, set the appropriate parameters for the constructor of the base class:

```
public ScoreGameObject() : base("JewelJamFont", Color.White, Alignment.Right) { }
```

"JewelJamFont" is the name of a font that you'll need to add to the project via the Pipeline Tool. You can take the *JewelJamFont.spritefont* file from our example files. If you want to use your own font instead, make sure to use the correct font name in your `ScoreGameObject` constructor!

The only other method that `ScoreGameObject` needs to override is `Update`. In that method, use the expression `JewelJam.GameWorld.Score` to get the current score. Convert that value to a string via the `ToString` method, and set it as the text:

```
public override void Update(GameTime gameTime)
{
    Text = JewelJam.GameWorld.Score.ToString();
}
```

That's all you need to do for this class. To see it in action, you need to add an instance of this class to the game world. Go to the `LoadContent` method of `JewelJamGameWorld` and add the following code:

```
// add a background sprite for the score object
SpriteGameObject scoreFrame = new SpriteGameObject("spr_scoreframe");
scoreFrame.LocalPosition = new Vector2(20, 20);
AddChild(scoreFrame);

// add the object that displays the score
ScoreGameObject scoreObject = new ScoreGameObject();
scoreObject.LocalPosition = new Vector2(270, 30);
AddChild(scoreObject);
```

This code first adds the background image on which the text will be drawn. It then draws the text on top of that image. Note: *spr_scoreframe* is another sprite that you'll have to add to the project.

Compile and run the game again. You'll now see the score in the top-left corner of the screen. Each time you create a valid combination of jewels, the displayed score will be increased by 10.

By the way, we could also have chosen to *not* write a separate `ScoreGameObject` class. Instead, we could have just used a standard `TextGameObject` to display the score, without any extra behavior attached to it. It would then become the task of `JewelJamGameWorld` to update this text whenever the score changes. This approach is also fine—it's up to you which version you prefer.

15.4.5 Extra Points for Multiple Combinations

As a fun addition to the game, we'll reward the player if he/she has created multiple valid combinations at the same time. This is quite easy to add. The idea is to not just add 10 points every time, but to make that number larger if there are multiple "hits" in one frame.

In the `HandleInput` method of `JewelGrid`, just before the **for** loop, define the points that should be added for the *first* valid combination:

```
int extraScore = 10;
```

Then, inside the loop, instead of always adding the value 10, add the current value of `extraScore`:

```
JewelJam.GameWorld.AddScore(extraScore);
```

and immediately after that, increase the value of `extraScore` for the rest of this method:

```
extraScore *= 2;
```

That way, a second combination will give the player 20 points, and a third combination will give 40 points. Note: this applies only for multiple combinations *within the same move*. The next time `HandleInput` gets called, it will start with an `extraScore` of 10 again, because `extraScore` is a local variable inside that method.

The score mechanism is now finished! Feel free to play around with the exact score values. If you want to reward a combo even more, why not multiply `extraScore` by 10 instead of 2? In the end, for your own games that you intend to publish, you'll have to balance such numbers very carefully. It's actually pretty hard to find the numbers that give the best experience for the player.

15.5 A Moving Jewel Cart

To give the player a feeling of pressure, we'll add a decreasing timer to the game. By creating valid combinations of jewels, the player can earn extra time.

Instead of drawing the actual remaining time, we'll add a jewel cart that slowly moves to the right. Earning extra time means that the cart gets pushed back to the left a bit. When the cart has left the screen, the game is over.

From a game programming perspective, it's a bit ugly to store a moving cart instead of an actual timer. This makes it difficult to determine how much time the player *really* has left. Still, we choose to do it like this for now, because otherwise the code would get too complicated for this phase of the book. In the last game of this book (in Part V), you'll see an example of a more "regular" timer.

15.5.1 The *JewelCart* Class

The behavior of the jewel cart is described in a new class, `JewelCart`. The code for this class can be found in Listing 15.3 and in the `JewelJam4d` example project. Here is an overview of the class:

- The constructor takes the jewel cart's starting position as a parameter. The *x*-coordinate of this starting position is stored as a backup.
- The `Reset` method sets the horizontal speed to 10 pixels per second, and it sets the cart's *x*-coordinate to the backup that was stored. That way, whenever the game is reset, the cart restarts at its original position and then starts moving to the right.
- The `PushBack` method pushes the cart 100 pixels back to the left (thus giving the player 10 extra seconds), while ensuring that the cart doesn't move past its starting position. This method should get called whenever the player scores points.
- All other data and behavior is already inherited from the `SpriteGameObject` class.

Add this class to your game now, and add the sprite *spr_jewelcart* to the project as well.

Listing 15.3 The JewelCart class

```

1  using Microsoft.Xna.Framework;
2
3  class JewelCart : SpriteGameObject
4  {
5      // The horizontal speed at which the cart moves.
6      const float speed = 10;
7
8      // The distance by which the cart will be pushed back if the player scores points.
9      const float pushDistance = 100;
10
11     // The x coordinate at which the jewel cart starts.
12     float startX;
13
14     public JewelCart(Vector2 startPosition)
15         : base("spr_jewelcart")
16     {
17         LocalPosition = startPosition;
18         startX = startPosition.X;
19     }
20
21     /// <summary>
22     /// Pushes the cart back by some distance, to give the player extra time.
23     /// Call this method when the player scores points.
24     /// </summary>
25     public void PushBack()
26     {
27         LocalPosition = new Vector2(
28             MathHelper.Max(LocalPosition.X - pushDistance, startX),
29             LocalPosition.Y);
30     }
31
32     public override void Reset()
33     {
34         velocity.X = speed;
35         LocalPosition = new Vector2(startX, LocalPosition.Y);
36     }
37 }

```

15.5.2 Adding the Cart to the Game

The game world should contain a single JewelCart instance. This instance should be part of the list of game objects, just like all other game objects. However, because we need to call the PushBack method sometimes, it's useful to store an *extra reference* to the JewelCart instance. For that reason, add another member variable to JewelJamGameWorld:

```
JewelCart jewelCart;
```

In the LoadContent method of JewelJamGameWorld, initialize the jewelCart variable and add it to the game world:

```
jewelCart = new JewelCart(new Vector2(410, 230));
AddChild(jewelCart);
```

And in the `AddScore` method, call `PushBack` using that `jewelCart` variable:

```
public void AddScore(int points)
{
    score += points;
    jewelCart.PushBack();
}
```

Note that the game world now stores *two* references to the *same* instance of `JewelCart`. One reference is given by the `jewelCart` member variable, and another reference is given by one of the items in the children list. Because the jewel cart is already part of the children list, it will already update itself in every frame (just like all other child objects). Therefore, you don't have to write another instruction `jewelCart.Update(gameTime);` yourself. With such an extra instruction, you would update the same jewel cart twice per frame, which would make the cart move twice as fast!

In other words, the `jewelCart` member variable is simply here so that we can easily call the `PushBack` method. Without this extra member variable, we would have to go through the entire children list and find the specific child object that is a `JewelCart`.

Extra References to Objects — In general, it's OK to store extra references to the objects that you need often, especially if they're difficult to find back otherwise. In this case, we've given the game world an extra `jewelCart` member variable that points to the same `JewelCart` instance that is also in the children list. We did this so that we can easily call `jewelCart.PushBack()`.

This idea is useful in many situations. For example, imagine a platform game with lots of enemies that should respond to the main character. (You'll actually create such a game in Part V of this book.) It's useful to give the enemies an easy way to access that character. Otherwise, all enemies will have to search for the character in each frame of the game loop. You can achieve this easy access in many different ways, such as with a read-only property in the game world, or with a member variable in the enemies themselves. It's a matter of taste which option you prefer.



CHECKPOINT: JewelJam4d

Compile and run the game again. You should now see the jewel cart slowly moving to the right. Whenever you score points in the game, the cart will move back a bit.

But in the current code, the jewel cart will keep moving after it has left the screen. What we'd like to do instead is show a "game over" screen that allows the player to reset the game, just in the Painter example. This is the topic of the next (and final) section.

15.6 Multiple Game States

In this section, you will extend the Jewel Jam game with different *game states*. This will allow you to change the game's appearance and behavior based on the current game state.

You may remember that the code of Painter contained a read-only property that returned whether or not the game is over. We used `if` instructions to let the `Update`, `HandleInput`, and `Draw` methods do different things based on that property. So, you could say that the Painter game had two different **game states**: "playing" and "game over."

The Jewel Jam game will have *four* game states instead of two:

- The game will start with a *title screen* that simply shows the title of the game.
- When the player presses the spacebar, the game moves to the *playing state*, which shows the actual game and allows player interaction.
- In the corner of the screen, we'll add a "Help" button. If the player clicks this button, the game goes to the *help state*, in which all objects are "frozen" and a help window is shown on top of everything else.
- If the jewel cart has left the screen, the game goes to the "*game over*" state. All regular interaction will be blocked again. By pressing the spacebar, the player can reset the game and go to the playing state again.

15.6.1 Adding Overlay Images

In some game states, we want to show a certain image on the foreground. An image that is drawn on top of everything else is also called an *overlay*. In Jewel Jam, there are three overlays: a title screen, a "game over" image, and a help screen. It's useful to prepare these images first, before we start implementing the game states themselves.

Start by adding three more sprites to the game: *spr_title*, *spr_gameover*, and *spr_frame_help*. For each sprite, we'll add a game object to the game world. Later, you'll want to make these objects visible or invisible depending on the game state. With that in mind, it's convenient to give the `JewelJamGameWorld` class three extra member variables:

```
SpriteGameObject titleScreen, gameOverScreen, helpScreen;
```

These will store convenient extra references to the overlay images, just like how you did that earlier for the jewel cart.

Eventually, we'd like to draw these overlay images in the center of the screen. To make this a bit easier, give the `SpriteGameObject` class the following method, which sets the origin of the object to the center of the sprite:

```
public void SetOriginToCenter()
{
    origin = new Vector2(Width / 2.0f, Height / 2.0f);
}
```

In the `LoadContent` method of `JewelJamGameWorld`, you can then add (for example) the title screen as follows:

```
titleScreen = new SpriteGameObject("spr_title");
titleScreen.SetOriginToCenter();
titleScreen.LocalPosition = new Vector2(Size.X / 2.0f, Size.Y / 2.0f);
AddChild(titleScreen);
```

Here, the first instruction creates the `SpriteGameObject` and stores it in the `titleScreen` member variable. The second instruction sets the object's origin. The third instruction sets the object's position to the center of the screen. The fourth instruction "officially" adds the object to the game world. This last line is important: if you leave it out, the overlays will never be drawn, because they're not part of the overall list of game objects.

Do the exact same thing for the help screen and the "game over" screen. If you don't want to copy and paste too much code, you can also create a helper method `AddOverlay` that combines these four

instructions. We’ve done this in our example code as well. Take a look at the JewelJam4e project if you’re curious.

If you compile and run the game now, you’ll see all images stacked on top of each other. Of course, this is not what we want: the images should become (in)visible when the game switches to a different *game state*. Let’s work on that feature next.

15.6.2 Using an Enum to Represent the Game State

Because there are now four possible game states (instead of two, such as in *Painter*), a single **bool** is no longer good enough to represent the game state. Instead, we can use an *enum* for it. In the *JewelJam* class, add the following enum and member variable:

```
enum GameState { TitleScreen, Playing, HelpScreen, GameOver }
GameState currentState;
```

The enum describes all the possible game states that exist. The member variable *currentState* will store the state that the game is *currently* in.

To switch to a certain game state, it’s convenient to give *JewelJamGameWorld* the following extra method:

```
void GoToState(GameState newState)
{
    currentState = newState;
    titleScreen.Visible = currentState == GameState.TitleScreen;
    helpScreen.Visible = currentState == GameState.HelpScreen;
    gameOverScreen.Visible = currentState == GameState.GameOver;
}
```

This method first sets the *currentGameState* member variable to the specified value. Next, it changes the visibility of the title screen, the help screen, and the “game over” image. For example, the title screen should only be visible when the state has been set to *GameState.TitleScreen*.

The syntax of the last three lines may look a bit weird (it includes both = and ==), but this is perfectly valid code. The code fragment *currentState == GameState.TitleScreen* is a Boolean expression, so it is **true** or **false**. The assignment operation (with the single =) passes that value to the *Visible* property.

15.6.3 Different Behaviors per Game State

The game should behave a bit differently in each game state. First of all, the game objects should only update themselves when the game is in the *Playing* state. You can achieve this by giving *JewelJamGameWorld* its own version of the *Update* method:

```
public override void Update(GameTime gameTime)
{
    if (currentState == GameState.Playing)
    {
        base.Update(gameTime);
        ...
    }
}
```

This makes sure that **base**.Update will *not* be called if the game is in any other state than Playing. That way, the game will be “frozen” in those game states.

At the position of “...”, we still need to add code that checks if the game is over. Remember that the game is over when the jewel cart has left the screen. So, as soon as the *x*-coordinate of the jewel cart has become big enough, we should go to the GameOver state. The following code does this:

```
if (jewelCart.GlobalPosition.X > Size.X - 230)
    GoToState(GameState.GameOver);
```

Remember to add this code *inside* the other **if** block, because we only want to check this when we’re in the Playing state. (By the way, the number 230 was chosen by us, so that the game is over when the cart has *just* left the screen. A large part of the jewel cart’s sprite consists of a “shine” effect, and we don’t want to include that part here.)

Next, the input handling should also be different depending on the game state. This means that JewelJamGameWorld needs to override HandleInput as well.

In the Playing state, the game should behave normally. In that case, all you need to do is call **base**.HandleInput:

```
if (currentState == GameState.Playing)
    base.HandleInput(inputHelper);
```

In the TitleScreen and GameOver states, you should check if the player presses the spacebar. When that happens, the game should reset itself, and the game should go to the Playing state:

```
else if (currentState == GameState.TitleScreen || currentState == GameState.GameOver)
{
    if (inputHelper.KeyPressed(Keys.Space))
    {
        Reset();
        GoToState(GameState.Playing);
    }
}
```

In the HelpScreen state, pressing the spacebar should also return the game to the Playing state. But in this case, the game should *not* reset itself, because we want the player to continue where he/she left off.

```
else if (currentState == GameState.HelpScreen)
{
    if (inputHelper.KeyPressed(Keys.Space))
        GoToState(GameState.Playing);
}
```

The game should start at the title screen. Therefore, at the end of LoadContent, replace the following instruction:

```
Reset();
```

by this:

```
GoToState(GameState.TitleScreen);
```

Later, the Reset method will be called as soon as the player presses the spacebar and the game begins.

Compile and run the game again. The game should now start at the title screen. Pressing the spacebar will start the game. If the jewel cart leaves the screen¹, the “game over” image gets shown and the game pauses. After that, pressing the spacebar will restart the game.

Game States in More Complex Games — The Jewel Jam game is still relatively easy: there are only a few different game states, and we know what the entire game should look like right from the start. Because of this, we can get away with loading the entire game world at the beginning and making some objects visible or invisible when the game state changes.

In larger games, things will be more complicated. For example, imagine a game with different *levels*, where each level is only loaded as soon as the player starts playing that level. Each level might even be a completely different type of game, such as in a collection of minigames. In such cases, a simple enum will no longer be enough to represent the game’s current state. Instead, each game state should show a completely different part of the game. It makes sense to then create a separate *class* for each game state, each with its own data and behavior. This also prevents you from having to store the entire game in memory all the time.

It looks like our current game state system will not be good enough for more complex games. As a result, we’ll have to revisit this topic later in this book.

15.6.4 Adding a Help Button

The last thing we need to do is add the “Help” button, which should set the game to the HelpScreen state when clicked.

Start by adding the sprite *spr_button_help* to the project. Add a *SpriteGameObject* with this sprite to the game world, and give it the position (1270, 20). This will draw the button in the top-right corner of the screen. Make sure to add the button *before* adding the overlays, so that the button is never drawn on top of an overlay. Furthermore, give *JewelJamGameWorld* a member variable *helpButton* that stores an extra reference to this button. This will make it easier to check if the button is being clicked.

Then, in the *HandleInput* method *JewelJamGameWorld*, we need to check if the player presses this button. We only need to check this when we’re in the *Playing* state. The button is pressed if the following two conditions hold:

1. The player presses the left mouse button.
2. The mouse position lies inside the rectangle that defines the help button.

Of course, the first condition is given by the following expression:

```
inputHelper.MouseLeftButtonPressed()
```

For the second condition, we can use the *BoundingBox* property that we added to the *SpriteGameObject* class some time ago. This property returns a *Rectangle*, which has a nice helper method named *Contains*. Given a *Vector2*, the *Contains* method checks whether the given vector lies inside your rectangle. So, it looks like the following expression checks whether the mouse pointer lies inside the help button:

```
helpButton.BoundingBox.Contains(inputHelper.MousePosition)
```

¹To test this more quickly, feel free to give the cart a much higher speed for now.

But there's one catch: the mouse position is given in *screen coordinates*, whereas the button itself has *world coordinates*. If you've forgotten about this difference, have a look at Sect. 12.4 again. To convert the mouse position to world coordinates, it would be nice if we could use the `ScreenToWorld` method of the `ExtendedGame` class.

Make that method **public**, so that other classes can access it. Unfortunately, even with this change, you cannot call `ScreenToWorld` from within the game world, because the `ScreenToWorld` method is *not static*. In other words, you need a specific instance of `ExtendedGame` to be able to call this method. We can't simply make the entire method static, because some of its instructions *also* require a specific instance of `ExtendedGame`.

There are multiple ways to fix this. We suggest the following solution:

- Give `JewelJamGameWorld` another member variable: `JewelJam game`. This will be an extra reference to the overall game, so that we can use the non-static methods of the `JewelJam` class.
- Give the constructor of `JewelJamGameWorld` a parameter: `JewelJam game`. At the beginning of the constructor, copy this parameter into the member variable:

```
this.game = game;
```

- Because the game world constructor has changed, the `JewelJam` class should now pass *itself* as a parameter when creating a `JewelJamGameWorld`:

```
gameWorld = new JewelJamGameWorld(this);
```

In the `JewelJamGameWorld` class, you can now write `game.ScreenToWorld` to call the `ScreenToWorld` method. With this solution, you can finally finish the `HandleInput` method. The complete code for the `Playing` state looks like this:

```
if (currentState == GameState.Playing)
{
    // handle the input for all game objects
    base.HandleInput(inputHelper);

    // if the player presses the Help button, go to the HelpScreen state
    if (inputHelper.MouseLeftButtonPressed() &&
        helpButton.BoundingBox.Contains(game.ScreenToWorld(inputHelper.MousePosition)))
    {
        GoToState(GameState.HelpScreen);
    }
}
```



CHECKPOINT: JewelJam4e

That's it: the gameplay for Jewel Jam is now completely finished. Compile and run the game again. You can now play Jewel Jam from start to finish, and you should be able to open and close the help window as well.

15.7 What You Have Learned

In this chapter, you have learned:

- how to program complicated game logic;
- how to implement the game-object interaction in a complex game world;
- how to distinguish between different game states, for example, to show a title screen.

15.8 Exercises

1. *Playing with a Grid*

In this question, you'll write the logic of a simple grid-based game. Take a look at the following class that contains a 2D array of integers:

```
class GameGrid
{
    int[,] numbers;
    public GameGrid(int width, int height)
    {
        numbers = new int[width,height];
    }
}
```

- Extend the constructor so that it fills the numbers array with random numbers between 1 and 5.
- Add a method `Increment` that takes a grid position (`Point pos`) as its parameter and that increases the number at that position by 1. If the result is 6, the number should be lowered to 1 again.
- Add a method `AllZeros` that checks whether the grid contains the number 0 in all grid cells. Use **return** to let the method stop as soon as you know the answer.
- * Add a method `RemovesIsland` that takes a grid position (`Point pos`) as its parameter. This method should replace the number n at that grid position by a 0. After that, all horizontal and vertical neighbors that store an n should *also* be replaced by 0. This process should continue until the entire “island” of connected n 's has been removed.
- * Embed this class in a game where the player can use the left mouse button to call `Increment` (on the grid cell that the mouse currently points at) and the right mouse button to call `RemovesIsland`. Make sure that cells with a 0 cannot be clicked. Keep track of how many clicks the player has made. The goal is to fill the grid with zeros in as few clicks as possible.

Chapter 16

Finishing the Game



In this chapter, you will add finishing touches to the Jewel Jam game: sound effects, music, and visual effects. For playing the sound effects and music, you'll add an extra class that handles all types of assets (including sprites and fonts). We'll get to that later in this chapter; we'll focus on the visual effects first.

The visual effects require some extra programming, but they're a good exercise. First, you will change the jewels so that they smoothly move to other positions in the grid. Second, you will show extra images when the player scores two or three combinations in one move (a "combo"). These images should hide themselves after a while again, which is another example of dealing with *time* in the game loop. Third, you will add glitter effects to the jewels and the combo images. To do that, you'll have to access the pixel data of sprites.

16.1 Making the Jewels Move Smoothly

Currently, whenever the `JewelGrid` class adds a jewel to a certain grid cell, it immediately gives that jewel the matching position in the game world. This makes the game a bit hard to follow: every change in the grid happens *immediately*, and the player can't really see what has happened exactly. To improve this, we'll add smooth movement to the jewels.

16.1.1 Adding a Target Position

A nice solution is to give each jewel a *target position*, which indicates the position that the jewel should move towards. First, give the `Jewel` class the following property:

```
public Vector2 TargetPosition { get; set; }
```

In the constructor of `Jewel`, initialize the target position to `Vector2.Zero`.

Next, we need to give the `Jewel` class a specific version of the `Update` method. In this method, a `Jewel` instance should make sure that it moves towards the target position. The nicest way to do this is to

change the *velocity* of the jewel. You can then call **base.Update** to do the rest of the work (i.e., updating the position according to the new velocity). The following code does the trick:

```
public override void Update(GameTime gameTime)
{
    Vector2 diff = TargetPosition - LocalPosition;
    velocity = diff * 8;

    base.Update(gameTime);
}
```

The first line of this method calculates the vector that goes from the current position to the target position. This vector is set as the velocity, scaled by a certain number to give a nice visual effect. (This is similar to what we did in *Painter*, for letting the mouse position determine how to shoot the ball.)

Note: if the difference between position and targetPosition is larger, then the velocity will also be larger. In other words, a jewel will move quickly if the target position is far away, and it will slow down as it gets closer to the target.

16.1.2 Setting the Target Positions of All Jewels

It's up to the *JewelGrid* class to set the target positions of all jewels. First, you need to give *JewelGrid* its own *Update* method. After all, the jewels didn't have any behavior so far, so there was no need to update them. But now, the *JewelGrid* is in charge of showing its children. Give this class an *Update* method that simply calls *Update* for all child objects:

```
public override void Update(GameTime gameTime)
{
    foreach (Jewel jewel in grid)
        jewel.Update(gameTime);
}
```

We also need to set the *TargetPosition* property of all jewels in the grid. Otherwise, all jewels will move towards the top-left corner of the screen, because *Vector2.Zero* is the default target position. (If you want to see this for fun, try running the game now.) To prevent that, we need to change a few more methods of *JewelGrid*.

Let's start by giving the jewels a correct target position when we *create* them. Our example projects already contained an *AddJewel* method that adds a new *Jewel* instance. Assuming you have such a method as well, you can add a single line that sets *TargetPosition* to the exact same value as *LocalPosition*:

```
void AddJewel(int x, int y)
{
    grid[x, y] = new Jewel();
    grid[x, y].Parent = this;
    grid[x, y].LocalPosition = GetCellPosition(x, y);
    grid[x, y].TargetPosition = GetCellPosition(x, y); // This line is new!
}
```

Next up, let's go to the methods that *move jewels around*. The *RemoveJewel* method removes a jewel from the grid, shifts a few jewels one row down, and then adds a new jewel at the top. In that method, look for the **for** loop that moves the jewels one row down. Change the code so that it sets the *TargetPosition* of a jewel instead of the *LocalPosition*. That way, you'll let the falling jewels move smoothly to their new positions, instead of immediately.

The `ShiftRowRight` method moves all jewels in a row one cell to the right and then reinserts the old rightmost jewel on the left side. In the **for** loop that replaces all jewels by their left neighbor, replace `LocalPosition` by `TargetPosition` again. This will make the jewels move smoothly to the right.

There's one jewel that "wraps around" the row, by disappearing on the right and reappearing on the left. For that jewel, set the `TargetPosition` to the first cell of the row. If you leave it at that, the jewel will move smoothly to the left, but it will move straight through the other jewels of that row. A slightly nicer effect is to let the jewel immediately "jump" to a position *to the left of the grid* and then let it move smoothly to its target cell. That way, all jewels will move at the same speed, and in the same direction.

You can use the `GetCellPosition` helper method to your advantage here: by giving that method an *x*-coordinate of `-1`, you'll get a world position that lies exactly one cell to the left of the grid. This is what the full `ShiftRowRight` method should look like:

```
public void ShiftRowRight(int selectedRow)
{
    Jewel last = grid[Width - 1, selectedRow];

    for (int x = Width - 1; x > 0; x--)
    {
        grid[x, selectedRow] = grid[x - 1, selectedRow];
        // The following line has changed!
        grid[x, selectedRow].TargetPosition = GetCellPosition(x, selectedRow);
    }

    grid[0, selectedRow] = last;
    // The following two lines have changed!
    last.LocalPosition = GetCellPosition(-1, selectedRow);
    last.TargetPosition = GetCellPosition(0, selectedRow);
}
```

In a similar way, try to change the `ShiftRowLeft` method yourself. Then run the game again to see if everything works correctly so far.

16.1.3 Letting the New Jewels Fall from the Sky

As a finishing touch, let's change the initial position of all new jewels that get created during the game. We'll let the jewels appear above the grid and then fall down smoothly.

To achieve that, give the `AddJewel` method another parameter:

```
void AddJewel(int x, int yTarget, int yStart)
```

Inside that method, *yTarget* is now the *y*-coordinate where the jewel should be added in the grid. Also, you should set the `LocalPosition` of the new jewel to match *yStart* and set the `TargetPosition` to match *yTarget*.

This will cause some errors in your code, because you have a few method calls to `AddJewel` that are currently missing a third parameter. In the `Reset` method, this parameter should simply be *y* again:

```
AddJewel(x, y, y);
```

That way, the jewels will already be at their resting position when the game (re)starts.

The `RemoveJewel` method also ends with a call to `AddJewel`. There, you could pass a *yStart* value of `-1`. This will let the new jewels spawn just above the grid. But that will look a bit strange, because we're always creating three new jewels at the same time. It would be nicer if one jewel spawns just

above the grid, one jewel spawns a bit higher, and one jewel spawns even higher. To achieve that, give the `RemoveJewel` method an extra parameter:

```
void RemoveJewel(int x, int y, int yStartForNewJewel)
```

and then pass this parameter to the `AddJewel` method:

```
AddJewel(new Jewel(), x, 0, yStartForNewJewel);
```

There are three calls to the `RemoveJewel` method, namely, in `HandleInput`. In those method calls, pass a parameter of `-1` for the first jewel, `-2` for the second jewel, and `-3` for the third jewel:

```
...
RemoveJewel(mid, y, -1);
RemoveJewel(mid, y+1, -2);
RemoveJewel(mid, y+2, -3);
...
```

This should give the desired “falling” effect.

By the way, if your code still contains the (unused) `MoveRowsDown` method, you’ll have to make some changes there as well. Our example projects don’t contain this method anymore, so we’ll leave this to you as an exercise.



CHECKPOINT: JewelJam5a

Compile and run the game again. All jewels should now move smoothly to their target positions in the grid.

As you can see, you had to change quite a lot of code to achieve this visual effect. Luckily, the compiler helps you out most of the time. If you give a method an extra parameter (like we’ve done `AddJewel` and `RemoveJewel`), you can “follow the errors” to see what else you need to change in your code.

16.2 Showing Combo Images for a Certain Amount of Time

When the player scores a double or triple combo (i.e., two or three valid combinations in a single move), we want to show a special image on the screen. This image should automatically disappear after a number of seconds.

Again, there are many ways in which you can do this. We’ll show you a solution that will also come in handy in other games: a separate game object that’s capable of changing the visibility of *another* object and that keeps track of the time that has passed.

16.2.1 Managing the Visibility of Another Object

Add another class `VisibilityTimer` to the `Engine` folder of your project. You can find the contents of this class in the `JewelJam5b` project, as well as in Listing 16.1. Here’s a quick overview of what the class does:

Listing 16.1 The `VisibilityTimer` class

```
1 using Microsoft.Xna.Framework;
2
3 /// <summary>
4 /// An object that can make another object visible for a certain amount of time.
5 /// </summary>
```

```

6 class VisibilityTimer : GameObject
7 {
8     GameObject target;
9     float timeLeft;
10
11     public VisibilityTimer(GameObject target)
12     {
13         timeLeft = 0;
14         this.target = target;
15     }
16
17     public override void Update(GameTime gameTime)
18     {
19         // if the timer has already passed earlier, don't do anything
20         if (timeLeft <= 0)
21             return;
22
23         // decrease the timer by the time that has passed since the last frame
24         timeLeft -= (float)gameTime.ElapsedGameTime.TotalSeconds;
25
26         // if enough time has passed, make the target object invisible
27         if (timeLeft <= 0)
28             target.Visible = false;
29     }
30
31     /// <summary>
32     /// Makes the target object visible, and starts a timer for the specified number of seconds.
33     /// </summary>
34     /// <param name="seconds">How long the target object should be visible.</param>
35     public void StartVisible(float seconds)
36     {
37         timeLeft = seconds;
38         target.Visible = true;
39     }
40 }

```

- When you create a `VisibilityTimer`, you have to supply a *target* object. This is a reference to another `GameObject` in your game.
- The `StartVisible` method makes that target object visible (via the `Visible` property). It also sets a member variable (`float timeLeft`) to a certain value.
- `VisibilityTimer` is a subclass of `GameObject`, which means that it inherits the `Update` method. In this method, it subtracts the elapsed time from the `timeLeft` member variable. When that timer has reached zero, `VisibilityTimer` makes the target object invisible again.

As you can see, you can quite easily use `MonoGame`'s `GameTime` class to create timers in your game. The main thing you need to do is keep track of the total time somewhere, and use the `gameTime` parameter of the `Update` method to change that value. You'll see this more often in the book.

16.2.2 Adding the Objects to the Game World

Next, let's add some more objects to the game world: two `SpriteGameObject` instances for the combo images themselves and two `VisibilityTimer` instances for the timers that control when the images are visible. Start by adding two new sprites to the project (*spr_double* and *spr_triple*).

We now need to add two `SpriteGameObject` instances at a certain position *and* attach a timer to them. Because we need to do exactly the same work for both combo images, it's useful to create a separate method for this. Add the following method to `JewelJamGameWorld`:

```
VisibilityTimer AddComboImageWithTimer(string spriteName)
{
    // create and add the image
    SpriteGameObject image = new SpriteGameObject(spriteName);
    image.Visible = false;
    image.LocalPosition = new Vector2(800, 400);
    AddChild(image);

    // create and add the timer, with that image as its target
    VisibilityTimer timer = new VisibilityTimer(image);
    AddChild(timer);

    return timer;
}
```

This method first creates a `SpriteGameObject` with the given sprite name and then adds it to the game world. It also makes sure that the image is initially invisible. Next, it creates and adds a `VisibilityTimer` that has the new `SpriteGameObject` as its target. Finally, it returns a reference to that timer, so that the game world can store it for convenience.

Add this method to `JewelJamGameWorld`. Because we'll want to activate these timers later on, it's convenient to store references to them in two new member variables:

```
VisibilityTimer timer_double, timer_triple;
```

With that in place, all you need to do is call the `AddComboImageWithTimer` method twice and store the result of those method calls in the two member variables you've just created. This should happen in the `LoadContent` method, of course:

```
timer_double = AddComboImageWithTimer("spr_double");
timer_triple = AddComboImageWithTimer("spr_triple");
```

Make sure to add these objects *after* the jewel cart but *before* the help window and other overlays.

16.2.3 Applying It to the Combo Images

The final step is to activate one of the two timers when the player scores a double or triple combo. For this, we suggest to give `JewelJamGameWorld` two extra methods:

```
public void DoubleComboScored()
{
    timer_double.StartVisible(3);
}
public void TripleComboScored()
{
    timer_triple.StartVisible(3);
}
```

These methods activate the appropriate timer to show a combo image for 3 s. The methods should be called whenever a combo is scored. The best place to do that is in the `HandleInput` method of `JewelGrid`,

because that method already checks for valid combinations and increases the score. In that method, introduce a new local variable, just before the **for** loop starts:

```
int combo = 0;
```

Whenever a valid combination is found, increase this combo counter by 1:

```
combo++;
```

And *after* the **for** loop, the combo variable stores how many combinations have been found in total. If this number is 2 or 3, then you want to call one of the new methods you’ve just added:

```
if (combo == 2)
    JewelJam.GameWorld.DoubleComboScored();
else if (combo == 3)
    JewelJam.GameWorld.TripleComboScored();
```

Again, remember that the combo variable will restart at 0 in every frame of the game loop. This is because it’s a *local* variable that exists only inside the `HandleInput` method, and not a *member* variable that “stays alive” when the method ends.



CHECKPOINT: JewelJam5b

Compile and run the game again. If you score a combo now, you should see one of the two combo images for a few seconds.

16.3 Adding Glitter Effects

To make the game look nicer, let’s add a glitter effect to the jewels and the jewel cart. We’ll do this via a new class, `GlitterField`, that is a subclass of `GameObject`. The `GlitterField` class is in charge of drawing a number of glitters on top of another sprite. In other words, it’s a single game object that manages a whole collection of glitters.

It would be a bit too intense to let you write this class step by step. Instead, go ahead and take the `GlitterField` class from the `JewelJam5c` example project. Also add the sprite *spr_glitter* to the project: this is the sprite of a single glitter in the field.

16.3.1 Outline of the *GlitterField* Class

Listing 16.2 The `GlitterField` class

```
1 using System.Collections.Generic;
2 using Microsoft.Xna.Framework;
3 using Microsoft.Xna.Framework.Graphics;
4
5 class GlitterField : GameObject
6 {
7     // The image of a single glitter
8     Texture2D glitter;
9     // The target image on which the glitter effect should be applied
10    Texture2D target;
11    // The rectangle within the target image that should receive glitters
```

```

12 Rectangle targetRectangle;
13
14 // The random positions of glitters in the field
15 List<Vector2> positions;
16 // The current scales of the glitters; these are numbers between 0 and 2
17 List<float> scales;
18
19 public GlitterField(Texture2D target, int numberOfGlitters, Rectangle targetRectangle)
20 {
21     // load the glitter sprite
22     glitter = ExtendedGame.ContentManager.Load<Texture2D>("spr_glitter");
23     // initialize some member variables
24     this.target = target;
25     this.targetRectangle = targetRectangle;
26     positions = new List<Vector2>();
27     scales = new List<float>();
28
29     // create random glitters
30     for (int i = 0; i < numberOfGlitters; i++)
31     {
32         positions.Add(CreateRandomPosition());
33         scales.Add(0f);
34     }
35 }
36
37 Vector2 CreateRandomPosition()
38 {
39     // keep trying random positions until a valid one is found
40     while (true)
41     {
42         // draw a random position within the target rectangle
43         Point randomPos = new Point(
44             ExtendedGame.Random.Next(targetRectangle.Width),
45             ExtendedGame.Random.Next(targetRectangle.Height)
46         ) + targetRectangle.Location;
47
48         // get the pixel data at that position
49         Rectangle rect = new Rectangle(randomPos, new Point(1, 1));
50         Color[] retrievedColor = new Color[1];
51         target.GetData(0, rect, retrievedColor, 0, 1);
52
53         // if the pixel is fully opaque, accept it as the answer
54         if (retrievedColor[0].A == 255)
55             return randomPos.ToVector2();
56     }
57 }
58
59 public override void Update(GameTime gameTime)
60 {
61     // update each glitter
62     for (int i = 0; i < positions.Count; i++)
63     {
64         // Let the glitter grow. If the glitter is currently invisible,
65         // it has a small chance to start growing.
66         if (scales[i] > 0 || ExtendedGame.Random.NextDouble() < 0.001)
67         {
68             scales[i] += 2 * (float)gameTime.ElapsedGameTime.TotalSeconds;
69             // If the glitter has reached scale 2, initialize a new random glitter.
70             if (scales[i] >= 2.0f)
71             {

```

```

72         scales[i] = 0f;
73         positions[i] = CreateRandomPosition();
74     }
75 }
76 }
77 }
78
79 public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
80 {
81     Vector2 glitterCenter = new Vector2(glitter.Width, glitter.Height) / 2;
82     for (int i = 0; i < scales.Count; i++)
83     {
84         float scale = scales[i];
85         // a scale between 1 and 2 means that the glitter is shrinking again
86         if (scales[i] > 1)
87             scale = 2 - scales[i];
88
89         // draw the glitter at its current scale
90         spriteBatch.Draw(glitter, GlobalPosition + positions[i], null,
91             Color.White, 0f, glitterCenter, scale, SpriteEffects.None, 0);
92     }
93 }
94 }

```

The source code for the `GlitterField` class can also be found in Listing 16.2. We will now give a quick overview of this class. To start off, it has a number of *member variables*:

- `glitter` is the sprite of a single glitter.
- `target` is the sprite on which the glitters should be drawn. `GlitterField` does not draw this sprite itself, because another game object already does that.
- `targetRectangle` is a rectangle within the target sprite. All glitters will be restricted to lie inside this rectangle.
- `positions` is a list of positions for the glitters that are currently shown.
- `scales` is a list of all glitter *scales*. Each glitter will start at scale 0 (invisible), then grow to scale 1 (full size), and then shrink back to 0. We'll represent this by a number that grows from 0 to 2. A "scale" between 1 and 2 will mean that the glitter is shrinking.

The *constructor* of `GlitterField` initializes all these member variables. One of the parameters of this constructor (`int numberOfGlitters`) determines how many glitters there are in total. The constructor fills the `positions` and `scales` lists with exactly this number of glitters.

The `Update` method increases each element of the `scales` list, based on the time that has passed in this frame. For each glitter that still has scale 0, there's a small chance that the glitter will start growing. We use the `Random` class for this, very similarly to what you've done earlier in the `Painter` game. If a scale has reached the number 2 (meaning that the glitter has completely shrunk again), that glitter will move to a new random position.

The `Draw` method simply draws all glitters at their current positions and scales. Any scales between 1 and 2 are properly converted, so that each glitter will first grow and then shrink. The scale is one of the nine parameters of `spriteBatch.Draw`.

16.3.2 Calculating a Random Glitter Position

The last method to discuss (and the most interesting one) is `CreateRandomPosition`. Roughly, this method calculates a random position within `targetRectangle`, but it only accepts positions of *nontrans*-

parent pixels in the target sprite. The result is that there will not be any glitters in the invisible parts of the target sprite, which looks more realistic.

The main **while** loop keeps trying random positions until an acceptable one is found. The instruction **while (true)** makes sure that the loop keeps running forever, until the program reaches a **return** instruction.

In the body of this loop, we first calculate a random pixel position within the target rectangle. You might not have seen the *Location* property before; this property returns the top-left corner of a *Rectangle*. Take a closer look at these lines until you have a good picture of what they do.

Next, we do something you haven't seen before: we get the *pixel information* of the target sprite. The *Texture2D* class has a method *GetData* that retrieves the colors of all individual pixels of the sprite. One of its parameters is an array of *Color* objects. If you give this method an array of the correct size, then this array will be filled in during the method call.

We use a version of *GetData* that has a *Rectangle* parameter, which only gets the pixel colors inside a certain rectangle of the sprite. In our case, we give the method a rectangle of 1 by 1 pixel, containing exactly the random pixel position we've just calculated. And because we're only interested in one pixel, we also give the method a *Color* array with only one element. (This looks a bit clumsy, but it's the only way in *MonoGame* to get the color of a single pixel.)

After the call to *GetData*, the *retrievedColor* variable stores an array, where index 0 stores the color of the requested pixel. A *Color* consists of four components: red, green, blue, and alpha. So far, we've only talked about the red, green, and blue components. The *alpha* component stores the **transparency** of the color. An alpha of 255 means that the color is fully visible (*opaque*), and an alpha of 0 means that the color is fully transparent. Any value in-between means that you can partly "see through" the color.

In this case, we check if the retrieved pixel has an alpha value of 255. In other words, we check if the pixel is fully opaque. If that's the case, then we accept this pixel as a glitter location. We convert the calculated *Point* to a *Vector2*, and then we return it.

16.3.3 Adding Glitter Fields to Game Objects

What remains is to add an *instance* of *GlitterField* to the game world. In this case, you'll want to add a glitter field to each *Jewel* instance, as well as to the *JewelCart* instance.

You can do this in many different ways. We suggest the following approach: give the *Jewel* and *JewelCart* classes their own *GlitterField* as a member variable, and let these classes be in charge of drawing and updating their own glitters. In other words, the glitter fields are not part of a *GameObjectList*, but they are simply "managed" by their owners. This approach keeps the game world nice and tidy.

Let's handle *JewelCart* first. Give that class the following member variable:

```
GlitterField glitters;
```

In the constructor of *JewelCart*, initialize it:

```
glitters = new GlitterField(sprite, 40, new Rectangle(275, 470, 430, 85));
```

This will create a field of 40 glitters in the specified rectangle. We've chosen this rectangle for you: it's the part of the jewel cart sprite that contains jewels. That way, players will only see glitters near the jewels themselves, and not all over the cart.

The *position* of the glitter field should depend on the position of the jewel cart. The easiest way to achieve this is to let the cart be the *parent object* of its own glitter field:


```
glitters.Parent = this;
```

Because a JewelCart now manages its own glitter field, it's also in charge of calling the Update and Draw methods for that glitter field. Thus, you need to give JewelCart its own overridden versions of the Update and Draw methods. The Update method should call **base**.Update and glitters.Update, and the same idea goes for the Draw method. You should be able to write those methods yourself now.

And that's all! Compile and run the game; you should now see random glitters on the moving jewel cart. The glitters will move along with the cart, thanks to the game-object hierarchy.

For the Jewel class, the work is mostly similar. Add a member variable of type GlitterField, initialize it in the constructor (we suggest to use 2 glitters instead of 40), and call the Update and Draw methods at the right moments. Also, don't forget to set **this** as the parent object of the glitter field. That way, the glitters will always move along with their jewel.

There's one important difference to JewelCart that you need to watch out for. Remember that we're using sprite sheet for all jewels combined, and each Jewel object only shows a *part* of that image. You need to give that part to the GlitterField constructor as well. Otherwise, the glitters will be based on a different image than what the jewel shows. In short, you need to initialize the glitter field as follows:

```
glitters = new GlitterField(sprite, 2, spriteRectangle);
```

where spriteRectangle is the member variable that Jewel already had, describing the part of the sprite to draw.

If you compile and run the game now, you should see one last problem: by accident, many glitters are shown to the right of the grid, and not on the jewels themselves. This is because the glitters all have positions inside spriteRectangle, but this rectangle may lie much farther to the right than the jewel itself. To compensate for that, add the following line to the Jewel constructor:

```
glitters.LocalPosition = -spriteRectangle.Location.ToVector2();
```

This gives the glitter field an *offset* that perfectly cancels out the “shifting” effect caused by spriteRectangle.



CHECKPOINT: JewelJam5c
The glitter effect is now finished!

Particle Systems — The glitter effect we've just added is an example of a **particle system**. A particle is a tiny object with its own behavior, and a particle system is an object that manages a lot of particles at the same time.

Particles are used a lot for special effects, both in games and in movies. They're an extremely useful way to visualize smoke, fire, explosions, magic spells, etc. Be careful: showing too many particles with complicated behavior may slow down your game. In the end, each particle is a sort of game object (but a very simple one).

16.4 Adding Music and Sound Effects

The final step is to add sound effects and music to the game. First, use the Pipeline Tool to import all songs and sound effects from our JewelJam example assets. This includes background music, a “game over” sound, and several sound effects for when the player scores points. Make sure that MonoGame treats the background music as a “Song” and the other sounds as a “Sound Effect.”

To *play* this sound and music, you could do exactly the same things as in the Painter game. However, for this chapter, we’ll make things a bit easier by adding a separate class that will handle assets for us.

16.4.1 An Improved Asset Manager

Add a class `AssetManager` to the “Engine” folder of your project. The idea is that this class will do *everything* related to loading sprites, fonts, sound effects, and songs.

Start by giving this class one member variable:

```
ContentManager contentManager;
```

This should be a reference to the `ContentManager` of the overall game. To fill in this reference, give the class a constructor that takes a `ContentManager` as a parameter:

```
public AssetManager(ContentManager content)
{
    contentManager = content;
}
```

Next, give the class four more methods, one for each of the asset types:

- `LoadSprite`, which loads and returns a *sprite* with a given name.
- `LoadFont`, which loads and returns a *font* with a given name.
- `PlaySoundEffect`, which loads a *sound effect* with a given name and immediately plays it.
- `PlaySong`, which loads a *song* with a given name and immediately plays it. Next to the name of the song, this method should have a second parameter that determines whether the song should loop around.

Try to write these methods yourself. Each method contains slightly different instructions, but you’ve seen those instructions before in previous projects. If you get stuck, you can find the answer in the `JewelJamFinal` example project.

Currently, the `ExtendedGame` class has the following property:

```
public static ContentManager ContentManager { get; private set; }
```

We did this so that other classes could load sprites directly. But now, we’d like our code to use the new `AssetManager` instead. Therefore, replace this property by the following:

```
public static AssetManager AssetManager { get; private set; }
```

This should cause an error in the constructor of `ExtendedGame`. There, you’re currently initializing the (old) `contentManager` variable:

```
ContentManager = Content;
```

Replace that instruction by the following:

```
AssetManager = new AssetManager(Content);
```

This will correctly create an instance of our new `AssetManager` class.

There should be a few more errors in your code, at the places where you're still using the old `ContentManager` property to load a sprite. This happens in the constructors of `SpriteGameObject`, `TextGameObject`, and `GlitterField`. Replace these incorrect instructions by method calls to `LoadSprite` and `LoadFont` of our new asset manager. Try to work out the details yourself or take a peek in the `JewelJamFinal` project.

16.4.2 Playing the New Sounds and Music

You've added five sound effects to the project. Most of them are related to scoring points: *snd_single*, *snd_double*, and *snd_triple* should be played when the player scores one, two, or three valid combinations at the same time. Furthermore, you can play *snd_error* when the player scores *zero* combinations. This will at least tell the player that he/she did something wrong, which is nicer than telling them nothing at all.

You can play these four sound effects in the `HandleInput` method of `JewelGrid`. After all, that method already has a nice local variable (`int combo`) that stores how many combinations have been found. At the end of that method, you're already using the `combo` variable to notify the game world of double and triple combos. Extend that code so that it plays different sound effects if `combo` has a value of 0, 1, 2, or 3. We won't give you the details here: try to write this code yourself.

The fifth sound effect, *snd_gameover*, should be played when the game goes to the "game over" state. This currently happens in the `Update` method of `JewelJamGameWorld`, when the jewel cart has left the screen. Extend that `if` block so that it also plays this sound effect.

And lastly, at the end of the `JewelJamGameWorld` constructor, play the background music:

```
ExtendedGame.AssetManager.PlaySong("snd_music", true);
```



CHECKPOINT: JewelJamFinal

This wraps up the Jewel Jam game. Compile and run the project to play the full version. Happy jewel hunting!

16.5 What You Have Learned

In this chapter, you have learned:

- how to give game objects more advanced behavior, by letting them move smoothly or by letting them disappear over time;
- how to access the pixel data of a `Texture2D` object, for example, to create glitter particles;
- how to group the handling of assets into a class;

This concludes Part III of the book. In this part, you’ve learned how to use arrays and collections to store many objects in a convenient way. You’ve also learned about recursion, a programming concept in which a method or property calls itself. Along the way, you’ve worked on a general object-oriented “game engine” that you can reuse in future games. This engine models the game world as a hierarchy of game objects. You have also learned how to deal with different screen sizes, which causes a difference between *screen* and *world* coordinates. These developments have led to the grid-based game Jewel Jam.

At this point, we can imagine that you’d like to extend the Jewel Jam game further. Feel free to add your own features to the game! In the exercises of this chapter, we’re already suggesting a few possible extensions.

In the next part of this book, you’ll work on a different game again, and you’ll improve your game engine even further. For example, you will learn how to read and write text files, how to add menus and settings to your game, and how to split a larger project into libraries. You will also learn about two important features of object-oriented programming: abstract classes and interfaces.

16.6 Exercises

1. *Extending the Jewel Jam Game*

Here are a few suggestions of things you could add to the Jewel Jam game, ordered from easy to difficult. Again, you’re free to implement these things however you want, and there usually is no such thing as a “best” answer. Good luck, and have fun!

- a. (*) The game is now quite easy: players can get far by simply bashing the arrow keys and the spacebar. To make things more interesting, give the player a penalty whenever there are *no* valid combinations if he/she presses the spacebar. You could give the player negative points, or you could push the jewel cart forward.
- b. (**) Give the player the option to give all jewels in the grid a new random color. This is useful if the player doesn’t see any valid combinations anymore. Of course, this action should come with a price: for example, you could give negative points again, or only allow a reset if the player has earned enough points since the last reset.
- c. (***) Add a fourth color of jewels to the game: a “rainbow” color. This color serves as a “wildcard” that you can interpret as any of the three other colors. This changes how you should check for valid combinations of jewels. Make sure that the rainbow jewels are very rare: don’t add them to the grid too often.
- d. (***) Add more *visual effects* for when the player scores a valid combination of jewels. For example, you could let the three jewels shrink or fly out of the screen. Note: this will probably mean that a grid of jewels alone is no longer enough. Some jewels will have to “stay alive” a bit longer without being part of the grid.
- e. (****) Give each jewel a small chance to contain a *power-up*. When a jewel with a power-up gets removed from the grid, a special bonus effect should occur. For example, an entire row or column gets cleared, all jewels of a certain type get removed, or the jewel cart gets pushed back by an extra distance. If a jewel contains a power-up, draw a small icon in the corner of that jewel, so that the player understands what’s going on.
- f. (*****) Add a *hint system* to the game. If the player hasn’t scored any combinations for some time, a “hint” button should appear. When the player presses this button, the game will automatically highlight three jewels in the grid that can form a valid combination. Note: these jewels may not be in the middle column yet! Finding such a combination automatically is pretty difficult. Think carefully about how you implement it.

2. Tic-Tac-Toe

With your knowledge of building Jewel Jam, you now have the tools and experience to create lots of other grid-based games. A simple example of such a game is Tic-Tac-Toe. This game features a 3×3 grid, and two players take turns to place a cross or a circle in that grid. A player wins if his or her symbol occurs three times in the same row, column, or diagonal.¹ This exercise helps you build this game step by step, but you'll need to figure out the details yourself. Try to reuse the “game engine” you’ve created so far, and always try to avoid duplicate code as much as possible.

- a. Start by looking up some nice sprites that you can use for this game. In any case, you’ll need a sprite for the cross and a sprite for the circle.²
- b. Make a first version of the game that only displays circles or crosses on a 3×3 grid on the screen. Here, you can reuse some of the classes that you’ve built for the Jewel Jam game.
- c. Next, add player input to the game. When the player clicks on the screen, a cross or circle should appear at that position in the grid. Make sure that players can only click on grid positions that are still free. Also, remember that this is a two-player game where players take turns. Therefore, you should keep track of whose turn it is, so that you can add the right element (cross or circle) to the grid.
- d. Next, write the code that checks whether there are three symbols of the same type in a row, column, or diagonal. You can do this by using a couple of **for** loops that check each of the possible combinations. Use this to determine if one of the players has won.
- e. Finally, handle the different game states: indicate on the screen whose turn it is, show an overlay if the game has been finished, and (if the game is over) allow the players to restart the game.
- f. * If you’re up for a challenge, try adding a game mode where a single player plays against the computer. You can start by letting the computer always choose a random non-empty position in the grid. After that, you can try to make the computer player more intelligent. Feel free to search the Internet for assistance!

3. * Other Grid-Based Games

Below are a few suggestions of other grid-based games that you can now create. For each game, we’ll give you a couple of hints to help you get started.

- a. **Minesweeper**: a single-player game featuring a grid with randomly placed mines. If the player clicks on a grid cell that contains a mine, the game is over. If the player clicks on an empty cell with at least one neighboring mine, that cell will show the number of neighboring mines. If the player clicks on an empty cell with no neighboring mines, an entire “island” of empty cells is revealed.
Hints: First calculate the number of neighboring mines for each grid cell, but don’t show it to the player yet. For revealing an “island” of empty cells, take a look at the exercise of Chap. 15.
- b. **Reversi**: a game where two players take turns to add a stone of their color to the grid. A player can “capture” the stones of the opponent by horizontally, vertically, or diagonally enclosing them by his/her own stones. If this happens, the captured stones will switch their color. The game continues until the entire grid is full. Then, the player with the most stones wins. (This game is sometimes also called *Othello*.)
Hint: The overall setup of this game (with players taking turns) is very similar to Tic-Tac-Toe. Only the game logic is very different.

¹In the Netherlands, where the authors of this book come from, this game is called “butter-cheese-and-eggs,” Nobody really knows why.

²You’re also allowed to use pictures of cheese and eggs.

- c. **Chess:** another turn-based game for two players, this time with different chess pieces that have different moves.

Hints: Create a subclass for each type of chess piece, and give them each their own version of a method that checks whether a move is valid. Can you add a hint functionality that shows all possible moves for a piece?

- d. **Tetris:** the famous puzzle game in which blocks fall down, and players need to stack these up nicely to keep the grid from overflowing. In this game, there's always one block that's falling down, and a grid of (possibly broken) blocks that have already fallen.

Hints: It's useful to create a separate Block class and to give that class methods for moving and rotating the block. Also consider adding subclasses for the different types of blocks that exists. Finally, it's useful to *not* immediately add the falling block to the grid. Only add the block to the grid when it has reached its stopping position.

Part IV

Menus and Levels

Introduction



In this part of the book, you're going to develop the game *Penguin Pairs*. This game has two main new aspects compared to *Jewel Jam*: separate menu screens and levels that can be loaded from text files. While implementing this game, you will learn several new techniques for game programming, such as sprite sheets, better game state management, file I/O, and more. You will also learn about a few important general programming concepts, such as abstract classes and interfaces.

About the Game: *Penguin Pairs* is a puzzle game in which the goal is to make pairs of penguins of the same color. The player can move a penguin by clicking on it and selecting the direction in which

the penguin should move. The penguin will move in the chosen direction until it is stopped by another object (this can be a penguin, a seal, a shark, or an iceberg). If there are no objects in the way at all, the penguin will fall off the playing field and into the icy water, where hungry sharks await.

Throughout the different levels of the game, we will introduce new gameplay elements to keep the game exciting. For example, there will be a special penguin that can match with penguins of any color, holes in which a penguin can get stuck, and sharks that are on the playing field itself.

To already play the final version of this game, you can open the solution belonging to Chap. 21 and run the *PenguinPairsFinal* project.

Chapter 17

Better Game State Management



In this chapter, you'll create the basis of the Penguin Pairs game. You will use the same game engine we've been creating so far, and you'll make a few improvements to it. These improvements are related to how the game deals with different *game states*.

In the Jewel Jam game, we used an enum to represent the current game state, and we simply made some objects (in)visible whenever the game state changed. That way, we could show a title screen when the game was first launched and a help window when the player pressed a certain button.

This setup is a bit too simple for Penguin Pairs: our new game will have a title screen, a menu with options, a help screen, a level selection screen, and the regular “playing state” itself. Some of these game states will be quite complicated, involving quite a lot of code and many game objects. Our code will get rather messy if we (again) create a single “game world” with the code for all different game states. It will also be a bit annoying if we have to make many objects (in)visible when the game state changes.

Can we do better than this? Well, we know that there's always only one game state active at the same time. Whenever a game state is *not* active, we would like to “skip” all the code related to that game state. For example, if the player is not currently looking at the menu, it would be a bit useless to update and draw all menu buttons.

In this chapter, you will create a way to *let each game state manage its own game objects* and to make sure that there's always exactly one active game state.

At the end of the chapter, you will have a system that manages different game states and keeps track of the current game state. Along the way, you will learn about *abstract classes* and *interfaces*, two very useful concepts in object-oriented programming.

17.1 New Classes for Game State Management

From now on, a **game state** is no longer just an enum value that says which state we're in. Instead, it will be an object that stores an entire “game world” of its own, with its own behavior. Our new system for game state management basically needs two things:

- There should only be one active game state at a time. For example, if the player is viewing the options menu, then only the “options menu” game state should update and draw itself.
- It should be easy to switch from one game state to another.

In this section, you’ll add two classes to the game engine: a `GameState` class that represents a single game state and a `GameStateManager` class that stores all game states and switches between them.

Start by creating a new MonoGame project. Give this project an “Engine” folder that contains all engine classes from Jewel Jam, such as `ExtendedGame`, `GameObject`, and `InputHelper`. Give the project the name “PenguinPairs,” and make sure that the main game class also has this name.

17.1.1 The *GameState* Class

Let’s add the class `GameState` to the engine. In essence, a game state is nothing more than a list of game objects, plus the usual game loop methods (`Update`, `Draw`, `HandleInput`, and `Reset`) that let these game objects do their work. The source code of the `GameState` class can be found in the `PenguinPairs1a` example project and in Listing 17.1. Go ahead and add this class to your Engine folder.

By the way, some game engines (such as Unity3D) use the term “scene” for an object that represents a single game state. We could have called this class `Scene` just as well.

Also, some programs or engines make a distinction between game *modes* and game *states*. In that case, a game mode is (for example) a menu or the main playing screen, whereas a game state is a simple enum value just like what we did in the Jewel Jam code. In other words, it’s possible to combine the two types of game state management that we’ve used so far. In this book, though, we’ll keep using our new `GameState` class from now on.

Listing 17.1 The `GameState` class

```

1 using Microsoft.Xna.Framework;
2 using Microsoft.Xna.Framework.Graphics;
3
4 class GameState
5 {
6     protected GameObjectList gameObjects;
7
8     protected GameState()
9     {
10         gameObjects = new GameObjectList();
11     }
12
13     public virtual void HandleInput(InputHelper inputHelper)
14     {
15         gameObjects.HandleInput(inputHelper);
16     }
17
18     public virtual void Update(GameTime gameTime)
19     {
20         gameObjects.Update(gameTime);
21     }
22
23     public virtual void Draw(GameTime gameTime, SpriteBatch spriteBatch)
24     {
25         gameObjects.Draw(gameTime, spriteBatch);
26     }
27
28     public virtual void Reset()
29     {
30         gameObjects.Reset();
31     }
32 }
```

17.1.2 A Collection of Game States

Next, let's add the class that manages all possible game states in the game and that keeps track of the current game state. The `GameStateManager` class will store a collection of `GameState` instances, plus an extra reference to the game state that's currently active. It will make sure that only the active game state updates and draws itself.

To make it easier to switch between states, we won't store the `GameState` in a list, but in a *dictionary*. We've briefly talked about dictionaries in Chap. 13, and now it's time to use one in practice! The `GameStateManager` class starts out like this:

```
using System.Collections.Generic;
class GameStateManager
{
    Dictionary<string, GameState> gameStates;
    GameState currentGameState;

    public GameStateManager()
    {
        gameStates = new Dictionary<string, GameState>();
        currentGameState = null;
    }
    ...
}
```

Remember from Chap. 13 that a `Dictionary` has two object types between its pointy brackets. The second type (after the comma) is the type of object that's actually stored (in this case `GameState`). The first type (before the comma) is the type of data by which these objects are *indexed* (in this case `string`). In this case, the strings are also called the *keys* of our dictionary, and the game states are the *values*.

The goal of this is that you can easily find a `GameState` in the collection by looking for a specific key. To make this work, give the `GameStateManager` class two extra methods:

```
public void AddGameState(string name, GameState state)
{
    gameStates[name] = state;
}

public GameState GetGameState(string name)
{
    if (gameStates.ContainsKey(name))
        return gameStates[name];
    return null;
}
```

The first method adds a `GameState` instance to the dictionary, stored under the key indicated by the `name` parameter. The second method retrieves the `GameState` that is stored under a given key. For safety, it first checks if the key really *exists* in the dictionary. If it does exist, it returns the game state with that key, using the squared brackets that you've also been using for arrays and lists. If the key does not exist, the method returns `null`, indicating that no game state was found.

17.1.3 Letting the Active Game State Do Its Job

Remember that our goal is to let only one game state be the "active" one and to ignore all other game states until they become active. We've already given the `GameStateManager` class a member variable

`currentGameState`, but this variable is initially **null**. To allow other classes to *change* the current game state, give `GameStateManager` the following method:

```
public void SwitchTo(string name)
{
    if (gameStates.ContainsKey(name))
        currentGameState = gameStates[name];
}
```

This method (again) looks for the game state with a particular key. If it was found, then that game state is set as the currently active state.

What remains is to add the four usual game loop methods (`Update`, `HandleInput`, `Draw`, and `Reset`). These methods should make sure that only the *current* game state does something and that the other game states do not. This is quite simple to add now. For example, here's the `Update` method:

```
public void Update(GameTime gameTime)
{
    if (currentGameState != null)
        currentGameState.Update(gameTime);
}
```

The other three methods work similarly, but they have different parameters. Try to write the rest of the code yourself—or take a look at the `PenguinPairs1a` example project if you're curious.

Loops and Dictionaries — You can also loop over the elements of a dictionary, using a **foreach** instruction just like with lists. However, in a dictionary, you're looping over special kinds of elements: each element is a *combination of a key and a value*. In C#, the data type for this is `KeyValuePair<A,B>`, where A and B should be replaced by the key and value types for your specific dictionary.

For example, a **foreach** loop over the `gameStates` dictionary looks like this:

```
foreach (KeyValuePair<string, GameState> pair in gameStates)
{
    ...
}
```

Given a `KeyValuePair` instance, you can use the `Key` property to retrieve the key (of type A) and the `Value` property to retrieve the value (of type B). For example, the following method finds and returns the key of the current game state. It does this by looping over the `gameStates` list and checking (for each key-value pair) if the value is equal to `currentGameState`. If so, it returns the corresponding key.

```
string GetCurrentGameStateKey()
{
    foreach (KeyValuePair<string, GameState> pair in gameStates)
    {
        if (pair.Value == currentGameState)
            return pair.Key;
    }
    return "";
}
```

17.1.4 Adding a *GameStateManager* to the Game

Finally, we need to change the `ExtendedGame` class so that it stores a `GameStateManager` instance. This will replace the old “game world” member variable that `ExtendedGame` currently has. After all, there is no longer a single game world, but a collection of possible “game worlds,” one of which is active.

So, go to the `ExtendedGame` class and change the following member variable:

```
protected static GameObjectList gameWorld;
```

into this:

```
public static GameStateManager GameStateManager { get; private set; }
```

By using a property with a public getter and a private setter, other classes can write `ExtendedGame.GameStateManager` to retrieve the game state manager.

Initialize this property in the `LoadContent` method. You’ll have to make some small changes in the `Update`, `Draw`, and `HandleInput` methods as well.

This concludes our preparations in the `Engine` folder.

17.2 Creating the Game States of Penguin Pairs

You can now use these two new classes to add the specific game states of the Penguin Pairs game. There are five game states in total, corresponding to the different screens that the player can see in the game: a title screen, an options menu, a help screen, a level selection menu, and a level itself.

For each of these game states, you’re going to add a new class that inherits from `GameState`. Each class will manage the objects and behavior for that specific game state.

17.2.1 Adding and Organizing the Assets

The example assets of this book include five background images, one for each game state: *spr_titlescreen*, *spr_background_options*, *spr_background_help*, *spr_background_levelselect*, and *spr_background_level*. Add these five images to your project via the Pipeline Tool. These are all the assets you will need for now.

Because this game will have quite a lot of assets eventually, we recommend placing the sprites in a folder named *Sprites*. Later, you’ll add other folders for the sounds and fonts. Figure 17.1 shows what the Pipeline Tool looks like after you’ve created a folder with these sprites and pressed Build.

17.2.2 Adding a Class for Each Game State

Next, add five new classes to the project: `TitleMenuState`, `OptionsMenuState`, `HelpState`, `LevelMenuState`, and `PlayingState`. To keep things organized, we recommend placing these classes in a new folder named *GameStates*. Figure 17.2 shows what the Visual Studio project should look like after this step. You can also find this structure in the `PenguinPairs1a` example project.

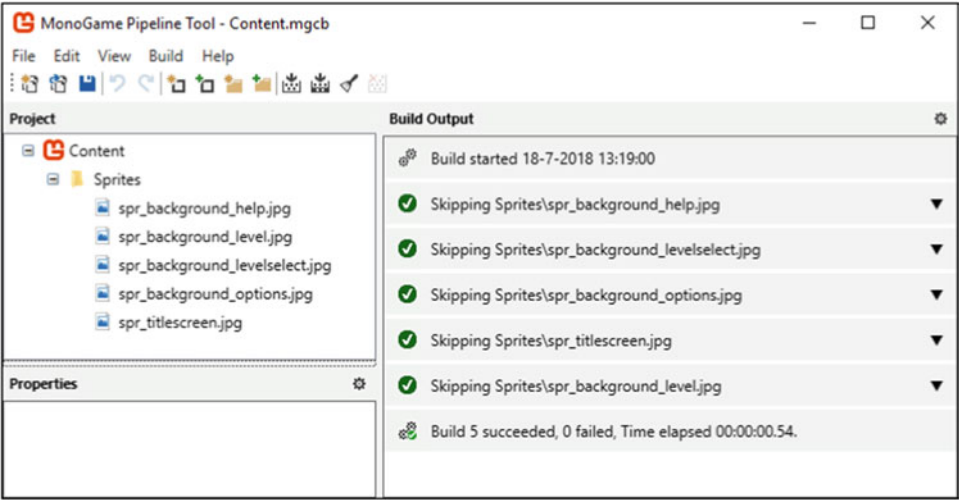


Fig. 17.1 A folder with sprites in the Pipeline Tool

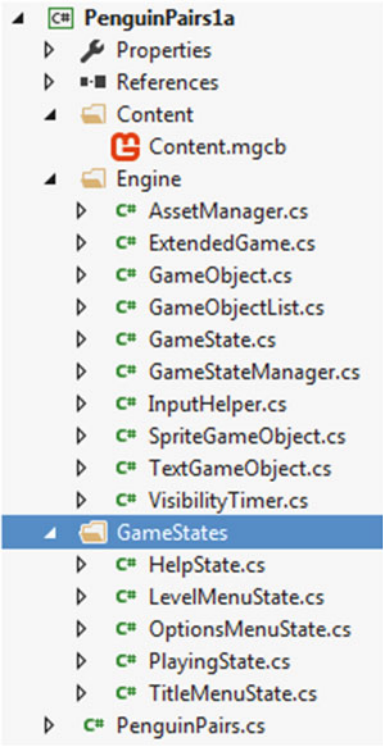


Fig. 17.2 The structure of the PenguinPairs1a example project

The code for each class is quite similar: each class should inherit from the `GameState` class, and the constructor method should add the correct background image to the game world. Let's take `TitleMenuState` as an example:

```
class TitleMenuState : GameState
{
    public TitleMenuState()
    {
        SpriteGameObject titleScreen = new SpriteGameObject("Sprites/spr_titlescreen");
        gameObjects.AddChild(titleScreen);
    }
}
```

Remember that `gameObjects` is a member variable of the `GameState` class. Because it is **protected**, subclasses of `GameState` can also access it.

Go ahead and write this code for all five classes, with a different background sprite for each class. You'll give the classes more code throughout the next chapters.

17.2.3 Preparing the *PenguinPairs* Class

Let's move on to the `PenguinPairs` class, the main class of our game. It should inherit from `ExtendedGame`, just like how you did it for `JewelJam`. This class should have the usual `Main` method and a constructor that makes the mouse pointer visible. Everything else is already handled by the parent class. So, this is how the `PenguinPairs` class starts out:

```
using System;
using Microsoft.Xna.Framework;

class PenguinPairs : ExtendedGame
{
    [STAThread]
    static void Main()
    {
        PenguinPairs game = new PenguinPairs();
        game.Run();
    }

    public PenguinPairs()
    {
        IsMouseVisible = true;
    }
    ...
}
```

(If your project still contains a *Program.cs* file with only a `Main` method, please remove that file now.)

The other method that you'll need to add is `LoadContent`. Just like in the `JewelJam` class, this method starts by calling **base**.`LoadContent` and setting the world size to 1200×900 :

```
protected override void LoadContent()
{
    base.LoadContent();

    worldSize = new Point(1200, 900);
    windowSize = new Point(1024, 768);
}
```



```
// to let these settings take effect, we need to set the FullScreen property again
FullScreen = false;
...
}
```

As soon as you call **base.LoadContent**, the parent class prepares a **GameStateManager** object, which you can access via the property **GameStateManager**. That part of the work is the same for all **ExtendedGame** instances.

At the end of **LoadContent**, you can now add the game states that are specific to **Penguin Pairs**. Remember that we've given **GameStateManager** an **AddState** method. This method takes a **string** as a key and a **GameState** as a value, and it stores that combination in a dictionary. So, the following line:

```
GameStateManager.AddGameState("titleScreen", new TitleMenuState());
```

will add a **TitleMenuState** object to the dictionary, using "titleScreen" as a name. In the same way, you can add the other four game states. You can choose their names yourself, as long as all five names are different (because the keys in a dictionary should be unique).

After you've added the five game state, you still need to specify at which state the game *starts*. You can do this via the **SwitchTo** method that you've added before. The following line will make sure that the game starts at the title screen:

```
GameStateManager.SwitchTo("titleScreen");
```

This concludes the **LoadContent** method. If you compile and run the game now, you should see the **Penguin Pairs** title screen. Note that there's no way for the player to switch to a different game state yet.

17.2.4 Adding Constants for Object Names

If you ever want to switch to a different game state, you'll have to give the correct string to the **SwitchTo** method. From a programmer's point of view, this is a bit dangerous. If you accidentally make a typo, the game will crash because your misspelled key doesn't exist. Also, if you ever want to *change* a game state's name because you're not happy with it, you'll have to go through your code to make the same change everywhere.

In the **PenguinPairs1a** example project, we've done something better: we've declared *constants* in the **PenguinPairs** class. At the top of the class, we've written the following:

```
public const string StateName_Title = "title";
public const string StateName_Help = "help";
public const string StateName_Options = "options";
public const string StateName_LevelSelect = "levelselect";
public const string StateName_Playing = "playing";
```

Everywhere else in the code, you can now use those constants instead of the actual strings. For example, the last line of **LoadContent** can now look like this:

```
GameStateManager.SwitchTo(StateName_Title);
```

If you use the constant names everywhere, the program will always use the strings that are "hidden" behind them. This is a much safer solution. Also, if you ever accidentally type a constant name that doesn't exist (such as **StateName_Titel**), the compiler will automatically tell you that something is wrong. We strongly recommend that you use this approach in your project as well.

17.2.5 Switching Between Game States

To allow the player to switch between game states, override the `HandleInput` method of each specific game state. In the next chapter, we'll add nice menus with buttons. For now, let's add some simple keyboard interaction:

- In the title screen, the player can press the “H” key to go to the help screen, “O” to go to the options menu, and “L” to go to the level selection screen.
- In the playing state, the player can press Backspace to go back to the level selection screen.
- In the other three game states, the player can press Backspace to go back to the title screen.

Try to write the code for this yourself. As an example, here is the `HandleInput` method of the `HelpState` class:

```
public override void HandleInput(InputHelper inputHelper)
{
    base.HandleInput(inputHelper);
    if (inputHelper.KeyPressed(Keys.Back))
        ExtendedGame.GameStateManager.SwitchTo(PenguinPairs.StateName_Title);
}
```

Note that the expression `ExtendedGame.GameStateManager` gives you access to the game state manager. Also, `PenguinPairs.StateName_Title` is one of the constants that we've added just now. And just as a reminder, both of these expressions are static, so they start with the *name* of the class they're in (and not a specific *instance* of that class).



CHECKPOINT: PenguinPairs1a

If you compile and run the game now, you should be able to switch between screens by pressing the correct keyboard keys.

Figure 17.3 shows an overview of the five game states and how they are related. The arrows indicate that the player can go from one state to another. Note: we have not yet implemented the transition from the level selection screen to the playing state, but we'll get to that later.

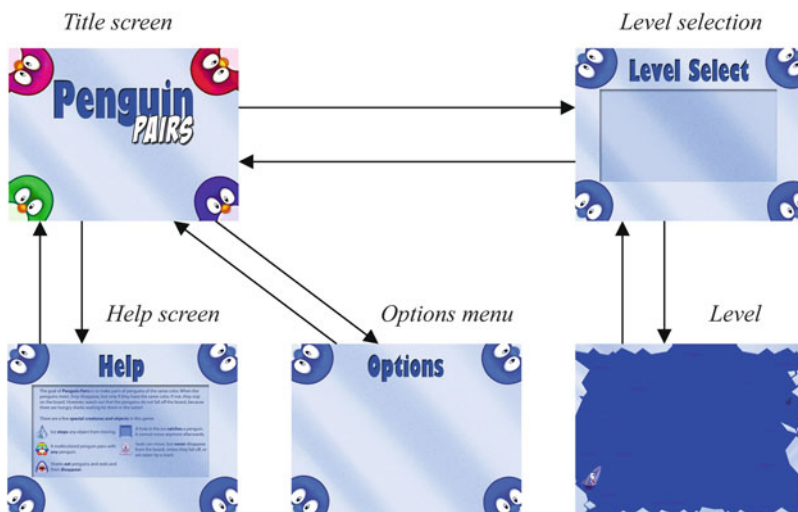


Fig. 17.3 The five game states of Penguin Pairs and their transitions

As soon as a game gets a bit more complicated, it's useful to draw diagrams like this one. This gives yourself (and possibly other members of your team) a good picture of the different components of the game.

In the next sections, you'll structure the program even better, using two new programming concepts: *abstract classes* and *interfaces*. You'll only make very small changes to the code, with very useful effects.

17.3 Abstract Classes

You've added five subclasses of `GameState`, one for each specific game state of Penguin Pairs. In the main `PenguinPairs` class, you've created a new instance of each game state.

But if you think about it, you'll never create an instance of the `GameState` class *itself*. You will always create a more specific version instead. After all, the `GameState` class is too "abstract": it defines only the standard behavior that you will need in every game state, and it doesn't define anything else. By creating subclasses of `GameState`, you can create versions that are specific for your game.

17.3.1 What Is an Abstract Class?

It would be nice if we could somehow tell the program that there can never be any instances of the `GameState` class itself. That way, it will be forbidden to write something like `new GameState()`, simply because we've promised ourselves that this is not allowed. C# actually has a nice feature for this: **abstract classes**.

To turn any class into an abstract class, simply add the word **abstract** to the class header. For example, for the `GameState` class, change the header into this:

```
abstract class GameState
```

From that point on, it will be forbidden to create a new instance of the `GameState` class. Try to write `new GameState()` somewhere (e.g., in the `LoadContent` method of `PenguinPairs`) and you'll see that the compiler doesn't allow it anymore. At the same time, you can still create instances of all *subclasses* of `GameState`, because those subclasses are *not* abstract.

When you see this for the first time, you may wonder why abstract classes are useful. Why can't programmers just pinky-promise that they won't create any `GameState` instances? Well, yes, that's also possible.¹ Abstract classes are not *required* in a program.

However, you can use abstract classes to make a program much *clearer* for other programmers. If a programmer sees the keyword **abstract**, they'll immediately know what they're dealing with: a class of which they cannot make new instances. Also, instead of having to check yourself that all programmers keep their promises, it's safer and easier to let the compiler take care of that.

By the way, abstract classes are not exclusive to C#: they're also supported by many other object-oriented languages, such as Java and C++.

¹ Although we're not sure if a pinky-promise has any legal value.

17.3.2 Abstract Classes in the Game Engine

Next to the `GameState` class, your game engine contains a few other classes that you can mark as **abstract**. These are the classes of which you'll *never* create any instances, but that you'll definitely need subclasses of. Can you guess which ones they are?

There are two of these classes in the `Engine` folder: `ExtendedGame` and `GameObject`. Every game will have a main class that inherits from `ExtendedGame`, but the main class will never be `ExtendedGame` itself. Also, each game object will either be a `SpriteGameObject`, a `TextGameObject`, or a `GameObjectList`. You will never create an instance of `GameObject` itself, simply because it doesn't provide enough functionality on its own.

In the `PenguinPairs1b` example project (and all project from this point on), we've made these classes abstract.

Way back in `Painter` game, the `ThreeColorGameObject` class was also a good candidate for becoming abstract. After all, we never created any instances of that class itself, but only of its subclasses (such as `Ball` and `PaintCan`).

17.3.3 Abstract Methods

You can go even further with abstract classes, though. Within an abstract class, you can also add **abstract methods**. Unlike the methods you've seen so far, an abstract method is a method *without a body*. By adding such a method to your class, you will promise that the method *exists* (and with which parameters), but you don't yet have to specify what it actually does. To live up to your promise, all subclasses of your abstract class *must* then implement this method.

This is useful if you know that multiple subclasses need to do the same type of task but that the *details* of this task will be complete different for each class. For example, imagine a game with the following abstract class:

```
abstract class Animal
{
    public void Speak();
}
```

This says that all animals in the game should be able to speak, without giving any further details. Any class that inherits from `Animal` must then implement the `Speak` method:

```
class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Woof!");
    }
}
```

Just like with normal inheritance, you use the keyword **override** to tell the compiler that this `Speak` method is indeed a more specific version of `Animal`'s `Speak` method. A (non-abstract) class is only valid if it implements *all* abstract methods of its parent class.

Of course, a child class of an abstract class can itself be abstract again. In that case, it doesn't yet have to implement all abstract methods.

Note: Abstract methods are only allowed inside abstract classes. In other words, a class with at least one abstract method *must* be an abstract class. However, an abstract class is also allowed to have zero abstract methods. The `GameState` class is currently an example of this.

In our game engine, we don't have to add any abstract methods yet. However, you now at least know about them, and we expect that you'll want to use them soon enough in games of your own.

17.3.4 Abstract Properties

Next to abstract *methods*, an abstract class can also have abstract *properties*. In that case, you only specify whether the property has **get** and **set** components, but you don't yet specify what those components do. Here are two examples:

```
protected abstract bool MyReadOnlyProperty { get; }
protected abstract int MyOtherProperty { get; set; }
```

And again, subclasses can fill in these abstract properties using the keyword **override**:

```
protected override bool MyReadOnlyProperty
{
    get { ... }
}
protected override int MyOtherProperty
{
    get { ... }
    set { ... }
}
```

You won't need abstract properties anytime soon in this book, but it's good to know that the possibility exists.



Quick Reference: Abstract Classes

If you add the keyword **abstract** to the header of a class `MyClass`:

```
abstract class MyClass { ... }
```

then that class will be *abstract*. This means that it's forbidden to create an instance of `MyClass` anywhere in your code (by writing `new MyClass(...)`). However, you're still allowed to create instances of *subclasses* of `MyClass`. This is useful if `MyClass` describes code shared by other classes, but doesn't describe a concrete object yet.

An abstract class can also have abstract *methods* (and properties). Such a method only has a header and no body:

```
public abstract void MyAbstractMethod();
```

Subclasses of `MyClass` can then implement this method, using the keyword **override** as usual:

```
public override void DoSomething() { ... }
```

If a non-abstract class `A` inherits from an abstract class `B`, then `A` is only valid if it implements *all* abstract methods and properties of `B`.

In programs with a big class hierarchy (such as the `Vehicle` hierarchy from Chap. 10), you'll often see that many classes are abstract. This is simply part of the overall program design. As you become a more experienced programmer, you will get a better feeling of when to use abstract classes and methods. Luckily, abstract classes are a safe topic to play around with: you only have to add a few keywords to your code. In other words, don't be scared to play around with this concept!

17.4 Interfaces

The `GameState` class contains a number of methods that are related to the game loop: `HandleInput`, `Update`, `Draw`, and `Reset`. The `GameObject` class has all of those methods as well. It would be nice if we could somehow make this similarity explicit in our code. For example, maybe we could put these four methods on a “checklist” and tell the compiler that both `GameState` and `GameObject` need to contain all methods from that checklist. That way, if we ever come up with a *fifth* method that both classes should have (let's say, a `Dance` method), we can simply add it to the checklist, and the compiler will automatically check if `GameState` and `GameObject` indeed contain a method named `Dance`.²

So far, you've seen one way to create such a checklist: you can create an *abstract class* (say, `GameLoopObject`) that contains all of these methods. You can mark all methods as **abstract** and leave their implementation for a subclass. If you turn `GameState` and `GameObject` into subclasses of `GameLoopObject`, then you force these classes to implement those methods.

This works, but there's one downside: a class can only have *one superclass*. What if you ever add a *second* checklist of methods (e.g., a `DancingObject` class, containing all the types of dances that objects can do)? The following class header is not allowed:

```
class GameObject : GameLoopObject, DancingObject
```

because you cannot give a class two superclasses. This problem is called *multiple inheritance*: you'd like a class to have multiple parent classes, but that's not allowed. It looks like abstract classes are not the ideal way to define a checklist of methods.

17.4.1 What Is an Interface?

Luckily, in C# (and in other languages as well), there's a second way to define a checklist of functionalities. Instead of creating an abstract class, you can also create an **interface**.

An interface is a group of methods and properties that only have headers and no body. If you tell the compiler that another class “implements” this interface, then that class must contain all of the interface's methods (and properties). In that sense, an interface is very similar to an abstract class with only abstract methods. But here's the trick: *a class can implement multiple interfaces*.

The header of an interface starts with the keyword **interface**, followed by a name you can choose yourself. It's common (but not mandatory) to let the name of an interface begin with a capital “I.” The body of an interface is a sequence of methods and properties without implementations, just like in a (completely) abstract class. Here's an example of an interface with one method:

```
interface IMoving
{
    void Move();
}
```

²Coming soon: *DiscoWorld 2!*

You can see a few more differences to abstract classes here. First, you don't have to write the word **abstract** anywhere. This is because an interface never contains any implementations anyway; you don't have to mention that twice. Second, the methods and properties of an interface cannot have an access modifier (such as **public** or **protected**). This is because the elements of an interface are *always public*.

17.4.2 Implementing an Interface

When you let a class implement an interface, you have to pay attention to a few things. Let's look at all the rules that are involved. It may look a bit intimidating, but it becomes easier if you remember this one-sentence summary: *an interface is a list of public methods (and properties) that every implementing class needs to include somehow*.

To tell the compiler that a class implements the methods of an interface, you need to add the name of that interface to your class header. For example, here's the header of a class that implements the IMoving interface:

```
class MyClass : IMoving
```

Note that we're using the word "implement" here, and not the word "inherit." It's common to use the word "inherit" when you're dealing with superclasses and "implement" when you're dealing with interfaces. But apart from that, the two concepts are quite similar.

If you promise that MyClass implements the IMoving interface, then this class should implement *all* methods (and properties) of that interface. In our running example, the compiler will check if MyClass contains all methods that were promised in IMoving. So, MyClass needs to contain at least a Move method:

```
public void Move()
{
    // do something
}
```

When implementing a method (or property) of an interface, you don't have to use the keyword **override**. That keyword is reserved for inheritance only. Also, you *must* make your method **public**; that's just a rule for interfaces that we need to follow.

By the way, you *are* still allowed to add the keyword **virtual** for methods that you implement:

```
public virtual void Move()
{
    // do something
}
```

This has the usual meaning: it allows you to create subclasses of MyClass that have a more specific version of the Move method. (The keyword **virtual** actually has nothing to do with implementing the interface.)

And as a final side note, if the class MyClass is **abstract**, then it doesn't *have* to implement all interface methods. You're then also allowed to just mention the method's header again (with the keyword **abstract**) and leave the actual work to subclasses.

17.4.3 Implementing Multiple Interfaces

As we've said before, the main advantage of interfaces is that a class can implement more than one interface. If you want to do that, you simply write multiple interface names behind the ":" symbol, separated by commas:

```
class MyClass : IMoving, IDancing, ISinging
```

A class is only valid if it mentions the methods and properties of *all* interfaces that it promises to implement. So, in this last example, MyClass would need to contain everything that is specified in IMoving, IDancing, and ISinging.

And finally, a class is allowed to implement interfaces *and* have a superclass at the same time. In the class header, behind the ":" symbol, you should write the name of the superclass first and then the names of the interfaces. For instance, if you want MyClass to also be a subclass of GameObject, you'd have to write the following header:

```
class MyClass : GameObject, IMoving, IDancing, ISinging
```

But enough about the rules. Let's apply this concept to our game engine!



Quick Reference: Interfaces

An interface is a "checklist" of methods or properties that other classes can implement. It's almost like a "light" version of an abstract class:

```
interface MyInterface
{
    void MyMethod1(bool x);
    int MyProperty { get; set; }
}
```

If you write the interface name in a class header:

```
class MyClass : MyInterface
```

then that class needs to contain all methods and properties that you defined in the interface. These methods all need to be **public**. This is also called *implementing* the interface.

A class can implement *multiple interfaces*, but it can only have *one superclass*. If a class has a superclass *and* implements one or more interfaces, you need to mention the superclass first:

```
class MyClass : MySuperclass, MyInterface, MyOtherInterface
```

17.4.4 The IGameLoopObject Interface

Let's add an interface to the game engine. In the PenguinPairs1b example project, we've added a file *IGameLoopObject.cs* (in the "Engine" folder). The contents of this file are shown in Listing 17.2.

Add this file to your project as well. Alternatively, you can also create a new file yourself: right-click on the “Engine” folder, choose Add → New Item → Interface, and give your new file the name *IGameLoopObject.cs*. Then replace the code in that file by the code shown in Listing 17.2.

Listing 17.2 The IGameLoopObject interface

```

1 using Microsoft.Xna.Framework;
2 using Microsoft.Xna.Framework.Graphics;
3
4 interface IGameLoopObject
5 {
6     void HandleInput(InputHelper inputHelper);
7     void Update(GameTime gameTime);
8     void Draw(GameTime gameTime, SpriteBatch spriteBatch);
9     void Reset();
10 }
```

The IGameLoopObject interface contains exactly the four game loop methods that we mentioned before. The next step is to tell the compiler that *GameObject* and *GameState* both implement this interface. This means that you should change their class headers:

```

abstract class GameObject : IGameLoopObject
abstract class GameState : IGameLoopObject
```

And that’s all you have to do! After all, *GameObject* and *GameState* already implement all four methods, so the compiler will be satisfied. To summarize, the IGameLoopObject interface is a “checklist” of methods related to the game loop, and you’re using it to promise the compiler that certain classes implement this checklist.



CHECKPOINT: PenguinPairs1b

You should now be able to compile and run the program again.

You might want to experiment some more, to get a better feeling of how interfaces work. You could try out the following things and see what kinds of errors the compiler gives:

- Remove one of IGameLoopObject’s methods from the *GameObject* class. The compiler will then tell you that the class doesn’t implement the entire interface anymore.
- Add a fifth method to the interface. If *GameObject* and *GameState* do not contain this method, the compiler will complain again (for the same reason).
- Mark one of these methods in *GameObject* as **private** or **protected**. The compiler will then say that all interface methods must be public.

As you can see, it’s very easy to add an interface to existing code. It’s a risk-free way to structure your program a bit better. You can also do it the other way around: define the interface first, and then design your classes around it. But usually, you discover later that your classes have a couple of methods in common. If it’s useful for your program to turn these shared methods into an explicit checklist, then an interface is nice to add.

17.4.5 Interfaces and Polymorphism

There’s another cool trick related to interfaces. You are now allowed to write something like this:

```
IGameLoopObject myObject = new PlayingState();
```

That is, if a class implements the interface `IGameLoopObject`, you can store it in a variable that has `IGameLoopObject` as its type! You can then also write the following:

```
myObject.Reset();
```

because the compiler knows that `myObject` definitely has a `Reset` method. It doesn't need to know what the method actually *does*; it only needs to know that the method *exists*. When the program runs, the computer will realize that `myObject` is actually an object of type `PlayingState`, and it will call the corresponding version of `Reset`. This is an example of **polymorphism**, which we've discussed a lot in Chap. 10. It works exactly the same with interfaces as it does with inheritance.

Likewise, you could even create a *collection* of `IGameLoopObject` instances:

```
List<IGameLoopObject> myList = new List<IGameLoopObject>();
myList.Add(new PlayingState());
myList.Add(new SpriteGameObject("Sprites/spr_titlescreen"));
...
```

and then loop over that list, for example, to call the `Reset` method for each element:

```
foreach (IGameLoopObject obj in myList)
    obj.Reset();
```

Just like with inheritance, your program will automatically call the correct *version* of `Reset` for each element. But for the compiler, it's enough to know that the method at least exists. The `IGameLoopObject` interface is a nice way to guarantee that.

By the way, the following line is *not* allowed:

```
IGameLoopObject myObject = new IGameLoopObject(); // Nope!
```

Just like an abstract class, an interface isn't a concrete type of object that you can directly create. After all, an interface is just a set of methods that *other* classes should implement.

Collections and Interfaces — Interfaces are used quite a lot in the standard classes of C#. In particular, the *collections* of C# (such as `List` and `Dictionary`) implement many different interfaces. Let's look at the `List` class, for example. The header of `List` looks like this:

```
public class List<T> : IList<T>, ICollection<T>,
    IList, ICollection, IReadOnlyList<T>,
    IReadOnlyCollection<T>, IEnumerable<T>, IEnumerable
```

In other words, the `List` class implements a whopping eight interfaces! Each interface has its own specific purpose. For example, the `IEnumerable` interface promises that you can loop over the elements of a class. The `ICollection` interface promises that a class contains the property `Count`. The `IReadOnlyList` interface promises that you can obtain the element at a certain index via square brackets.

Because `List<T>` implements `IList<T>`, you're also allowed to write the following:

```
IList<int> myList = new List<int>();
```

The header of the `Collection` class looks even crazier! Feel free to look it up if you're feeling adventurous. You don't have to understand all the details here— but it's at least fun to know that the standard code C# itself is full of interfaces.

17.5 What You Have Learned

In this chapter, you have learned:

- how to manage different game states between which the player can switch;
- how to use abstract classes and interfaces to give your program an even better structure.

17.6 Exercises

1. *Abstract Classes and Interfaces (1)*

What is the difference between an abstract class and an interface? Give an example of a situation in which you would use an abstract class. Give another example of a situation in which you would use an interface.

2. *Abstract Classes and Interfaces (2)*

Below are a few statements about abstract classes and interfaces. Which of them are true, and which of them are false?

- An abstract class must have at least one abstract method.
- An abstract class cannot have any private methods.
- An abstract method can never be private.
- The methods of an abstract class can never have a body.
- The methods of an interface can never have a body.
- A class can inherit from only one class but from multiple interfaces.

3. *Abstract Classes: What Is Allowed?*

Given are the following classes:

```
abstract class A
{
    public abstract void Method1();
    public void Method2() { }
}
class B : A
{
    public override void Method1() { }
    public void Method3(A a) { a.Method1(); }
}
```

Indicate for each of the following instructions whether or not it is allowed:

```
A obj;
obj = new A();
obj = new B();
obj.Method1();
obj.Method2();
obj.Method3(obj);
B otherObject = (B)(new A());
A yetAnotherObject = (A)obj;
obj.Method3(otherObject);
A[] list;
list = new A[10];
list[0] = new A();
list[1] = new B();
List<A> otherList = new List<A>();
```

4. *Loading Game States Dynamically*

Our example code fills the `GameStateManager` at the very beginning of the game. If you have a game with many game states, this may be a bit too much. It would be nicer if you could add a game state *as soon as you need it*. For example, there's no need to load the full options menu if the player never opens it.

How can you change the code to make this possible?

5. *Lists and Interfaces*

For this question, you should know that the `List` class implements an interface `ICollection`.

- a. Below are three lines of code, each combining a declaration and an assignment. Only one of these lines is correct. Which one is that, and why are the other two *incorrect*?

```
List<int> a = new ICollection<int>(); // version 1
ICollection<int> b = new List<int>(); // version 2
ICollection<int> c = new List<int>(); // version 3
```

- b. * By the way, the following line is also correct:

```
List<int> d = new List<int>(); // version 4
```

Describe a situation in which version 1/2/3 (the correct line that you chose in the previous subquestion) has an advantage over version 4.

Chapter 18

User Interfaces and Menus



In the Jewel Jam game, you’ve already seen a basic “Help” button in the top-right corner. The Penguin Pairs game will contain similar objects, for example, in the options menu. There, the player can change two game settings: the music volume and the availability of in-game hints.

Elements such as buttons, sliders, and switches form the so-called *user interface* (UI) of your game. In this chapter, you’ll add new classes for UI elements to the game engine. You’ll use those classes to add buttons and sliders to the relevant game states of Penguin Pairs. At the end of the chapter, you will also fill the level selection screen in which the player can choose a level to play.

But first, let’s work on a general way to let a single image file contain multiple sprites. This will be useful for some of our UI elements but also for the penguins in the game itself.

18.1 Sprite Sheets

In Jewel Jam, we used a single image that contained all 27 different jewel sprites. This was a nice way to reduce the total number of images that the program needs to load. For Penguin Pairs, some of the example sprites work in a similar way. For example, Fig. 18.1 shows how all possible penguins (and a seal) are stored in a single image file. But this time, it’s not a single row of sprites, but a “grid” of sprites with multiple columns *and* multiple rows. Such an image is called a **sprite sheet**. Sprite sheets are used in many games.

A sprite sheet with only one row is sometimes called a *sprite strip*. But if a strip contains many sprites, it’s nicer to organize them in a sheet instead. In extreme cases, one very long strip could even become too wide for the graphics card to handle!

In this section, you’re going to write a `SpriteSheet` class that handles sprite sheets in a general way. This will make your game engine even more reusable.

18.1.1 Overview of the *SpriteSheet* Class

Add a `SpriteSheet` class to the Engine folder. Each instance of this class will represent a single sprite sheet. The class will contain a `Texture2D` member variable that stores the entire image and a `Rectangle` member variable that stores the part of the image that should be drawn. This is quite similar to what we did earlier for the `Jewel` class. In fact, the `SpriteSheet` class will be a more general version of what we did for `Jewel`.



Fig. 18.1 An example of a sprite sheet, with four columns and two rows

We'll design the `SpriteSheet` class in such a way that it can represent regular sprites as well. A regular sprite will simply be treated as a sprite sheet with one row and one column. That way, sprites and sprite sheets can be handled in exactly the same way. Eventually, we'll change the `SpriteGameObject` class so that it stores an instance of a `SpriteSheet`, instead of directly storing a `Texture2D`.

Listing 18.1 shows an incomplete version of the `SpriteSheet` class. Go ahead and use this as a starting point. Let's take a moment to understand this code.

The class contains the `Texture2D` and `Rectangle` member variables that we mentioned before. It also contains three `int` member variables: these represent the number of rows and columns in the sprite sheet and the *index* of the sprite to use. For example, in the 4×2 sprite sheet of Fig. 18.1, index 0 corresponds to the top-left image (the blue penguin), index 3 corresponds to the top-right image (the yellow penguin), and index 6 corresponds to the third image on the second line (the rainbow penguin).

After the member variables, you see a set of read-only properties. The `Width` property returns the width of a single sprite in the sheet. This is simply the width of the entire image, divided by the number of columns in the sprite sheet. The `Height` property works similarly. These properties are useful because `SpriteGameObject` will sometimes ask for the width and height of its sprite. If that sprite is actually a whole *sheet* of sprites, then we don't want to return the width and height of the entire image, but only of the part that we're actually using. These `Width` and `Height` properties do exactly that.

There are also properties for returning the *center* and the *bounds* of a single sprite in the sheet. For completeness, we've also added a property that returns the total *number* of sprites in the sheet.

The final part that you see in Listing 18.1 is a custom `Draw` method. Because the standard `spriteBatch.Draw` method can only draw objects of type `Texture2D`, the `SpriteSheet` class needs to provide its own version of that method. As you can see, this `Draw` method simply calls `spriteBatch.Draw` again, but it uses the `spriteRectangle` member variable to only draw a certain part of the sprite sheet.

Listing 18.1 An incomplete first version of the `SpriteSheet` class

```

1 using Microsoft.Xna.Framework;
2 using Microsoft.Xna.Framework.Graphics;
3
4 class SpriteSheet
5 {
6     Texture2D sprite;
7     Rectangle spriteRectangle;
8     int sheetIndex, sheetColumns, sheetRows;
9
10    public int Width { get { return sprite.Width / sheetColumns; } }
11    public int Height { get { return sprite.Height / sheetRows; } }
12
13    public Vector2 Center
14    {
15        get { return new Vector2(Width, Height) / 2; }
16    }
17 }
```

```

16 }
17 public Rectangle Bounds
18 {
19     get { return new Rectangle(0, 0, Width, Height); }
20 }
21 public int NumberOfSheetElements
22 {
23     get { return sheetColumns * sheetRows; }
24 }
25 public void Draw(SpriteBatch spriteBatch, Vector2 position, Vector2 origin)
26 {
27     spriteBatch.Draw(sprite, position, spriteRectangle, Color.White,
28         0.0f, origin, 1.0f, SpriteEffects.None, 0.0f);
29 }
30 }

```

18.1.2 Adding a Constructor

We still need to give this class a *constructor* that initializes all member variables. Let's create a constructor with four parameters: the name of the image, the number of columns and rows in the sprite sheet, and the sheet index to use. The header looks like this:

```
public SpriteSheet(string assetname, int columns, int rows, int sheetIndex=0)
```

As briefly explained in Chap. 7, the “=0” part indicates a *default value* for the parameter `sheetIndex`. If you call this constructor, you're allowed to leave out the last parameter, and then a value of 0 will be assumed. It's useful to choose a default sheet index of 0 (indicating the top-left sprite of the sheet). That way, if your sprite is not a sheet at all, but just a single image, you can leave out the fourth parameter and everything will work fine.

Inside the constructor, you can start by loading the sprite and by setting the **int** member variables to the given parameter values:

```

sprite = ExtendedGame.AssetManager.LoadSprite(assetname);
sheetColumns = columns;
sheetRows = rows;
this.sheetIndex = sheetIndex;

```

After that, it's time to calculate the `Rectangle` that represents the part of the sprite to draw. This is (again) similar to what we did for `Jewel`, except that we can now have sprite sheets with multiple rows.

The width and height of the rectangle are always the same: they don't depend on the sheet index. In fact, they're already given by our `Width` and `Height` properties. The only thing that can be different is the *top-left corner* of the rectangle, which *does* depend on the sheet index. To calculate this top-left corner, it's useful to first calculate the row and column that we're looking for. The following instruction calculates the *column index*:

```
int columnIndex = sheetIndex % sheetColumns;
```

For example, if the sheet has four columns, a `sheetIndex` of 0 refers to the top-left sprite. An index of 1 refers to the second sprite and so on. A `sheetIndex` of 4 refers to the first sprite on the next row: from that point on, the pattern continues. Therefore, the expression `sheetIndex % 4` (using the modulo operator) gives us the column we need.

The following instruction calculates the *row index*:

```
int rowIndex = sheetIndex / sheetColumns;
```

For example, if the sheet has four columns, the numbers 0–3 indicate the first row, the numbers 4–7 indicate the second row, and so on. Therefore, the expression `sheetIndex / 4` gives us the correct row index. Remember: if you divide an integer by an integer, the result will automatically be rounded down.

Once you’ve found these two indices, you can turn them into pixel coordinates by multiplying them by `Width` or `Height`. You’ve seen this before in the `Jewel` class, too. The resulting two numbers represent the top-left corner of `spriteRectangle`:

```
spriteRectangle = new Rectangle(columnIndex * Width, rowIndex * Height, Width, Height);
```

This concludes the `SpriteSheet` constructor—or at least our first version of it.

18.1.3 Reading the Number of Sprites from the Filename

If you think about it, it’s a shame that we always have to specify the number of rows and columns in a sprite sheet. For a slightly nicer version of the constructor, we’ll add the following trick: the *name of the image file* will indicate how many rows and columns there are. We will implement three possibilities:

- If the image is a sprite sheet of $W \times H$ sprites, then the sprite name will end with “@WxH”. For example, the sprite sheet from Fig. 18.1 is called “spr_penguin@4x2,” indicating that the sprite sheet is 4 sprites wide and 2 sprites high.
- If the image is a sprite strip of W sprites, then the sprite name will end with “@W”. For example, if we’d put all sprites from Fig. 18.1 in one row, we could call it “spr_penguin8.”
- If the image is just a single sprite, then there is no special naming rule.

Our updated `SpriteSheet` constructor no longer needs parameters for the number of rows and columns. Instead, it will extract these numbers from the name of the image. Again, the constructor starts by loading the sprite. After that, it first assumes that there is only one row and column:

```
sheetColumns = 1;
sheetRows = 1;
```

Next, we check if the filename says that we’re dealing with a sprite sheet. To do this, we use an interesting method from the `String` class: `Split`. This method splits a string into parts, by cutting it at the places where a certain character occurs. The following line:

```
string[] assetSplit = assetname.Split('@');
```

splits the string `assetName` into parts that were separated by the “@” character. This “@” character will be removed from all parts. The result is an *array* of strings, with one array element for each separate part. For instance, if `assetName` has the value “spr_penguin@4x2”, then the result will be an array of two elements. The first element will contain the string “spr_penguin” and the second element will contain “4x2”.

If the resulting `assetSplit` array has only one element, then there was no “@” in the string at all, so we don’t have to do anything special. If the array has at least two elements, then we’re only interested

in the *last* element: that's the part that can contain something like "4x2". So, our constructor should continue like this:

```
if (assetSplit.Length >= 2)
{
    string sheetNrData = assetSplit[assetSplit.Length - 1];
    ...
}
```

At "...", we now have two options: either `sheetNrData` is a single number or it consists of two numbers with an "x" in between. We can start by using the `Split` method again, this time with the "x" character:

```
string[] columnAndRow = sheetNrData.Split('x');
```

The resulting array can have one or two elements. In both cases, the first element definitely indicates the number of columns:

```
sheetColumns = int.Parse(columnAndRow[0]);
```

And if there are two elements, the second element indicates the number of rows:

```
if (columnAndRow.Length == 2)
    sheetRows = int.Parse(columnAndRow[1]);
```

This concludes the main `if` block. After that, `sheetRows` and `sheetColumns` have been set, and you can calculate the sprite rectangle as usual. Use this description to write the full constructor yourself.

18.1.4 A Property for the Sheet Index

Let's add one final thing to the `SpriteSheet` class: we'll make it possible to *change* the sheet index during the program. This will be very useful in future projects.

To enable this, add a property `SheetIndex`. The **get** part of this property should simply return the value of the member variable `sheetIndex`. The **set** part of this property should first check if the new sheet index (given by the keyword **value**) is valid: it needs to be at least zero and smaller than the total number of sprites in the sheet. If this check succeeds, then we update the `sheetIndex` member variable. After that, we also have to recalculate `spriteRectangle`, because that rectangle depends on the sheet index.

In total, this is what the `SheetIndex` property should look like:

```
public int SheetIndex
{
    get { return sheetIndex; }
    set
    {
        if (value >= 0 && value < NumberOfSheetElements)
        {
            sheetIndex = value;

            int columnIndex = sheetIndex % sheetColumns;
            int rowIndex = sheetIndex / sheetColumns;
            spriteRectangle = new Rectangle(
                columnIndex * Width, rowIndex * Height, Width, Height);
        }
    }
}
```

The code that calculates a new `spriteRectangle` was already in the constructor, too. You can now remove that code there and replace it with a single instruction that calls the setter of `SheetIndex`. Make sure to do this at the *end* of the constructor: otherwise, the class hasn't calculated yet how many rows and columns the sprite sheet has!

Take a look at the `PenguinPairs2a` project to see our final version of the `SpriteSheet` class.

18.1.5 Updating the *SpriteGameObject* Class

Now that the `SpriteSheet` class is complete, the final step is to update `SpriteGameObject`. Instead of having a `Texture2D` member variable, that class should now store a `SpriteSheet`:

```
protected SpriteSheet sprite;
```

Basically, all parts of the class that used `Texture2D` should now use `SpriteSheet` instead. For example, the `Draw` method can no longer directly call `spriteBatch.Draw`. Instead, it should call the `Draw` method of the sprite sheet:

```
public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
{
    if (Visible)
        sprite.Draw(spriteBatch, GlobalPosition, origin);
}
```

Finally, it's convenient to give `SpriteGameObject` a public property `SheetIndex` that simply gets or sets the `SheetIndex` of the attached `SpriteSheet`. Try to add this property yourself.

That's it: you've now added sprite sheets to the game engine! You should be able to compile and run the game again now.

18.2 Menu Elements

Let's move on to the next topic: adding menus. The menus of `Penguin Pairs` will contain three important elements: buttons, switches, and sliders. Because these elements are so common in games, you'll start by writing general classes for them. These classes will be part of your game engine again.

In this section, we'll let you write these three classes one by one. In our example project, we've given the `Engine` folder a subfolder named `"UI,"` and all three classes are placed inside this folder. The term `"UI"` is short for **user interface**, a term that's often used for all menu-like structures that are shown on the screen. The buttons, sliders, and other elements of a UI are often called `"UI elements."` Sometimes, a UI is also called a GUI (short for *"graphical user interface"*), to make extra clear that the UI elements are drawn on the screen.

Watch out: the term `"user interface"` has nothing to do with the keyword **interface** in C#!

18.2.1 Obtaining the Mouse Position in World Coordinates

All of our UI elements will need to respond to mouse clicks in some way. Because the menu elements themselves are objects in the game world, they should be able to retrieve the mouse position in *world coordinates*.

Currently, the `InputHelper` class can only give the mouse position in *screen coordinates*. The `ExtendedGame` class has a `ScreenToWorld` method that converts screen coordinates to world coordinates. However, `InputHelper` can't access that method yet: `ScreenToWorld` is not static, so we need an *instance* of `ExtendedGame` before we can call the method.

We had the same problem before in *Jewel Jam*. Let's solve it in the same way now:

- Give the `InputHelper` class a custom constructor that takes an `ExtendedGame` object as a parameter. Inside the constructor, store a reference to that parameter in a member variable named `game`.
- Rename the property `MousePosition` to `MousePositionScreen`.
- Add a property `MousePositionWorld` that converts `MousePositionScreen` to world coordinates and returns the result. Use the game member variable for this.
- In the `ExtendedGame` class, when you initialize the `InputHelper`, give **this** as a parameter to the constructor.

This will allow objects to ask for the mouse position in both types of coordinates. Try to make these changes yourself, or take them from the *PenguinPairs2a* project.

18.2.2 The *Button* Class

The `Button` class, which represents a clickable button, is very simple. The full code is given in [Listing 18.2](#). Go ahead and add it to the `UI` folder of your project.

In summary, a button is a special type of `SpriteGameObject` that keeps track of whether it's being pressed. This is stored in a property `bool Pressed`.

Initially, a button is not pressed. In the `HandleInput` method, `Pressed` is set to **true** if the button is visible *and* the left mouse button is being pressed *and* the mouse pointer lies inside the button's bounding box. Otherwise, `Pressed` is set to **false**.

Other objects should be able to ask for the value of the `pressed` variable, but only the button itself (or future subclasses of `Button`) should be able to change it. Therefore, the **set** part of the `Pressed` property is **protected**.

Listing 18.2 The `Button` class

```

1 class Button : SpriteGameObject
2 {
3     public bool Pressed { get; protected set; }
4
5     public Button(string assetName) : base(assetName)
6     {
7         Pressed = false;
8     }
9
10    public override void HandleInput(InputHelper inputHelper)
11    {
12        Pressed = Visible && inputHelper.MouseLeftButtonPressed()
13            && BoundingBox.Contains(inputHelper.MousePositionWorld);
14    }
15
16    public override void Reset()
17    {
18        base.Reset();
19        Pressed = false;
20    }
21 }
```

18.2.3 The *Switch* Class

The *Switch* class is meant for representing an “on/off” switch, or a checkbox that can be checked and unchecked. In other words, a switch is basically a button, with the extra feature that it has a status (“on” or “off”) that changes each time the player presses the switch.

Our code for the *Switch* class can be found in Listing 18.3. *Switch* is a subclass of *Button*, so it inherits all button behavior. Apart from that, it has the following extras:

- A member variable that says whether or not the switch is currently on. This variable is initially **false**.
- A property (**bool** *Selected*) with a **get** part that returns the value of that member variable.
- The *HandleInput* method always *switches* the value of the *Selected* property when the switch gets pressed.
- The **set** part of the *Selected* property also gives the object a different *sprite sheet index*. This allows us to give a *Switch* object a sprite sheet with two sprites: one for the unselected switch (the “off” state) and one for the selected switch (the “on” state). Whenever the state of the switch changes, the appropriate sprite gets chosen. Figure 18.2 shows our sprite sheet for a switch in *Penguin Pairs*. It looks like a button with the word “on” or “off,” depending on the state of the switch. (This image is named “spr_switch@2”, to indicate that it’s a sprite sheet with one row of two sprites.)

Listing 18.3 The *Switch* class

```

1 class Switch : Button
2 {
3     bool selected;
4     public bool Selected
5     {
6         get { return selected; }
7         set
8         {
9             selected = value;
10            if (selected)
11                SheetIndex = 1;
12            else
13                SheetIndex = 0;
14        }
15    }
16
17    public Switch(string assetName) : base(assetName)
18    {
19        Selected = false;
20    }
21
22    public override void HandleInput(InputHelper inputHelper)
23    {
24        base.HandleInput(inputHelper);
25        if (Pressed)
26            Selected = !Selected;
27    }
28
29    public override void Reset()
30    {
31        base.Reset();
32        Selected = false;
33    }
34 }
```



Fig. 18.2 The sprite sheet for a switch in Penguin Pairs

18.2.4 The *Slider* Class

The final (and most complicated) UI element to add is the *slider*. Writing this class all by yourself is a bit tough, so in this section, we'll show you *our* version and explain our reasoning behind it. You can find the source code in Listing 18.4 and in the PenguinPairs2a example project. We encourage you to try and write your own version of the `Slider` class, using the following text as a guideline. If you get stuck or intimidated, feel free to use our code instead. Remember: there's not one "best" way to write this class, so your own version might be quite different from ours.

Let's start by thinking about the features that a slider should contain:

1. It should consist of two images: a wide bar in the background and a smaller block in the foreground that the player can move horizontally.
2. The position of the foreground block should represent a numeric value. When adding a slider to a menu, we should specify what the range of possible numbers is. This range is given by a minimum value (for when the foreground block is on the left end of the bar) and a maximum value (for when it is on the right end).
3. When the player is *holding* the left mouse button anywhere inside the bar, the foreground block should move to that position. The player doesn't have to be *pressing* the mouse button in that frame, because it should be possible to drag the foreground block around. The new position of the mouse determines the new value of the slider.
4. It should also be possible to set the slider's value from outside the class. When that happens, the foreground block should move to the correct position.
5. Other classes should be able to ask whether the slider has received a different value in the last frame. That way, you can trigger changes in the game when the player moves the slider around.

How do these items translate to code? Item 1 suggests that a `Slider` object has two sprites inside it. This means that we can let `Slider` extend the `GameObjectList` class and give it two `SpriteGameObject` instances as children. The constructor of `Slider` should take the names of these two sprites as its parameters.

Item 2 suggests that `Slider` needs three member variables of type `float`: the minimum value, the maximum value, and the *current* value of the slider. The constructor of `Slider` needs the minimum and maximum values as parameters. For the default "current" value, we can start at the minimum.

With the information so far, you can write the constructor and member variables of `Slider`. In our own version, we've added one extra feature: a padding member variable, which represents the thickness of the background image's border, in pixels. For example, our background image has a border of 8 pixels wide. To make sure that the foreground block stays nicely within this border, we need to take these 8 pixels into account when moving the block around. We will use this padding later in the class.

There's another detail worth mentioning. To make sure that the foreground block can reach the far left and right of the background bar, we need to change the *origin* of the foreground block. Up until now, `SpriteGameObject` only had a method `SetOriginToCenter`, but we now want to make sure that objects can receive *any* kind of origin. So, this is a good time to turn the origin of a `SpriteGameObject` into a more accessible property:

```
public Vector2 Origin { get; set; }
```

Please make this change now, and use this new property in the `Slider` constructor.

Item 3 suggests that we need to give the `InputHandler` class an extra method that returns whether the left mouse button is *currently down*, without caring about the previous frame. Go ahead and add this method now; give it the name `MouseLeftButtonDown`.

Converting the mouse position to a new slider value is rather tricky. In our version of the code, we've added a couple of read-only properties to make these calculations a bit easier. The `Range` property indicates the range of values that the slider can represent. `MinimumLocalX` and `MaximumLocalX` are the smallest and largest *x*-coordinate that the foreground block can have. `AvailableWidth` is the difference between these two: it's the total amount of movement space that the foreground block has. Note that these properties take the padding into account as well.

After these preparations, you can write the `HandleInput` method of the `Slider` class. If the slider is visible, this method first checks if the left mouse button is down and if the mouse pointer is inside the slider. If so, then the slider's value should change. To do this, our version of `HandleInput` first translates the *x*-coordinate of the mouse to a number between 0 and 1. This number is 0 when the mouse is on the far left of the slider and 1 when it is on the far right. Next, it converts that number to a value that lies within the slider's range. Take your time to understand how this code works.

Item 4 suggests that we're better off storing the slider's current value in a *public property*, where the **set** part takes care of moving the foreground block to the correct position. In our version of this property, the **set** part first stores the given number as the slider's new value. Next, it calculates the new position of the foreground block, by doing the exact opposite of what `HandleInput` does: it scales the new value to a number between 0 and 1 and then converts that to a pixel position for the foreground block.

Listing 18.4 The Slider class

```

1  using Microsoft.Xna.Framework;
2
3  class Slider : GameObjectList
4  {
5      // The sprites for the background and foreground of this slider.
6      SpriteGameObject back, front;
7
8      // The minimum and maximum value associated to this slider.
9      float minValue, maxValue;
10     // The value that the slider is currently storing.
11     float currentValue;
12
13     // The number of pixels that the foreground block should stay away from the border.
14     float padding;
15
16     public Slider(string backgroundSprite, string foregroundSprite,
17         float minValue, float maxValue, float padding)
18     {
19         // add the background image
20         back = new SpriteGameObject(backgroundSprite);
21         AddChild(back);
22
23         // add the foreground image, with a custom origin
24         front = new SpriteGameObject(foregroundSprite);
25         front.Origin = new Vector2(front.Width / 2, 0);
26         AddChild(front);
27
28         // store the other values
29         this.minValue = minValue;
30         this.maxValue = maxValue;
31         this.padding = padding;

```

```

32
33     // by default, start at the minimum value
34     Value = this.minValue;
35 }
36
37 // The difference between the minimum and maximum value that the slider can store.
38 float Range { get { return maxValue - minValue; } }
39
40 // The smallest X coordinate that the front image may have.
41 float MinimumLocalX { get { return padding + front.Width / 2; } }
42 // The largest X coordinate that the front image may have.
43 float MaximumLocalX { get { return back.Width - padding - front.Width / 2; } }
44 // The total pixel width that is available for the front image.
45 float AvailableWidth { get { return MaximumLocalX - MinimumLocalX; } }
46
47 public override void HandleInput(InputHelper inputHelper)
48 {
49     base.HandleInput(inputHelper);
50
51     if (!Visible)
52         return;
53
54     Vector2 mousePos = inputHelper.MousePositionWorld;
55     if (inputHelper.MouseLeftButtonDown() && back.BoundingBox.Contains(mousePos))
56     {
57         // translate the mouse position to a number between 0 (left) and 1 (right)
58         float correctedX = mousePos.X - GlobalPosition.X - MinimumLocalX;
59         float newFraction = correctedX / AvailableWidth;
60         // convert that to a new slider value
61         Value = newFraction * Range + minValue;
62     }
63 }
64
65 /// <summary>
66 /// Gets or sets the current numeric value that's stored in this slider.
67 /// When you set this value, the foreground image will move to the correct position.
68 /// </summary>
69 public float Value
70 {
71     get { return currentValue; }
72     set
73     {
74         // store the value
75         currentValue = MathHelper.Clamp(value, minValue, maxValue);
76         // calculate the new position of the foreground image
77         float fraction = (currentValue - minValue) / Range;
78         float newX = MinimumLocalX + fraction * AvailableWidth;
79         front.LocalPosition = new Vector2(newX, padding);
80     }
81 }
82 }

```

Finally, item 5 from our list suggests that `Slider` also needs to keep track of the value that it had in the *previous* frame. If you keep track of this value correctly, you can add a read-only property `bool ValueChanged` that returns whether the two values are different. Go ahead and add this feature yourself. It's not included in the listing here, but you *can* find it in the `PenguinPairs2a` example project.



CHECKPOINT: PenguinPairs2a

Once you’ve completed the Slider class, the UI part of the game engine is finished!

18.3 Adding the First Menus of Penguin Pairs

You’re now ready to add UI elements to the different game states of Penguin Pairs. Let’s start by adding new sprites to the project. In the Pipeline Tool, create a “UI” folder inside the “Sprites” folder, and fill it with all sprites from the “UI” folder of our example assets. Do the same for the “Fonts” folder as well: we’ll be drawing text on the screen very soon.

In total, the Pipeline Tool should now contain a “Fonts” folder with three fonts, a “Sprites” folder with the background sprites, and a “UI” folder inside the “Sprites” folder with all UI images. Now that the sprites and fonts are in place, let’s fill the different game states one by one.

18.3.1 Filling the Title Screen

The TitleMenuState class should create three buttons, one for each game state that is reachable from this screen. Start by giving this class three more member variables:

```
Button playButton, optionsButton, helpButton;
```

In the constructor, add these buttons one by one (after the background). For example, add the “Play” button as follows:

```
playButton = new Button("Sprites/UI/spr_button_play");
playButton.LocalPosition = new Vector2(415, 540);
gameObjects.AddChild(playButton);
```

In a similar way, add the “Options” button at position (415, 650), and add the “Help” button at position (415, 760).

In the HandleInput method of TitleMenuState, you’re currently changing the game state when the player presses certain keys on the keyboard. Now that you’ve added buttons, you can replace these keyboard presses by button presses. For example, to go to the level selection menu, the player should now press the “Play” button instead of the “L” key:

```
if (playButton.Pressed)
    ExtendedGame.GameStateManager.SwitchTo(PenguinPairs.StateName_LevelSelect);
```

Change the other two key presses into button presses as well. Once you’ve done that, you can compile and run the game again. When you click one of the buttons in the menu screen, you’ll go to the corresponding game state. You can still press the Backspace key to go back to the main menu.

18.3.2 Adding the Three Back Buttons

The HelpState, LevelMenuState, and OptionsMenuState classes should all receive a “Back” button at the bottom of the screen. Give each of these classes a new member variable:

```
Button backButton;
```


and initialize that variable as follows:

```
backButton = new Button("Sprites/UI/spr_button_back");  
backButton.localPosition = new Vector2(415, 720);  
gameObjects.AddChild(backButton);
```

In the `HandleInput` method, you should now check if `backButton` is pressed (instead of checking the Backspace key). Make these changes for all three classes.

18.3.3 Adding Buttons to the Playing State

The `PlayingState` class (which will show one playable level at a time) is a bit different. It should contain three buttons at the top of the screen: a “hint” button that can show a hint, a “retry” button that can restart the level, and a “quit” button that can send the player back to the level selection screen. For convenience, give the `PlayingState` class three member variables:

```
Button hintButton, retryButton, quitButton;
```

Add the “quit” button at position (1058, 20). Add the “hint” and “retry” buttons both at position (916, 20), and make the “retry” button invisible for now. In the next chapter, you’ll add code that makes the “retry” button visible as soon as the player has made a move, but don’t worry about that now.

Change the `HandleInput` method so that the game returns to the level selection screen when the “quit” button is pressed.

After you’ve made these changes, compile and run the game again. The buttons should now replace all keyboard input that the game used to have. The options menu and the level selection screen are still incomplete, though. (As a result, you cannot actually select a level yet, so you cannot see if the `PlayingState` class works correctly.) These two screens are both complicated enough to deserve a section of their own.

18.4 Filling the Options Menu

In this section, you’ll fill in the options menu of *Penguin Pairs*. You’ll also make sure that the options in this menu are linked to properties of the game.

18.4.1 Adding the UI Elements

Next to the “Back” button, the options menu will contain two more UI elements: a *switch* that the player can use to enable and disable hints in the game and a *slider* that can be used to change the volume of the background music. For both UI elements, we’ll also add a piece of text that explains what the UI element does.

Figure 18.3 shows what the full options menu should look like. Try to fill the constructor of `OptionsMenuState` so that it creates this menu. You’ll need to add a `Button`, a `Slider`, and two `TextGameObject` instances that use the *MenuFont* asset. It’s useful to add separate member variables

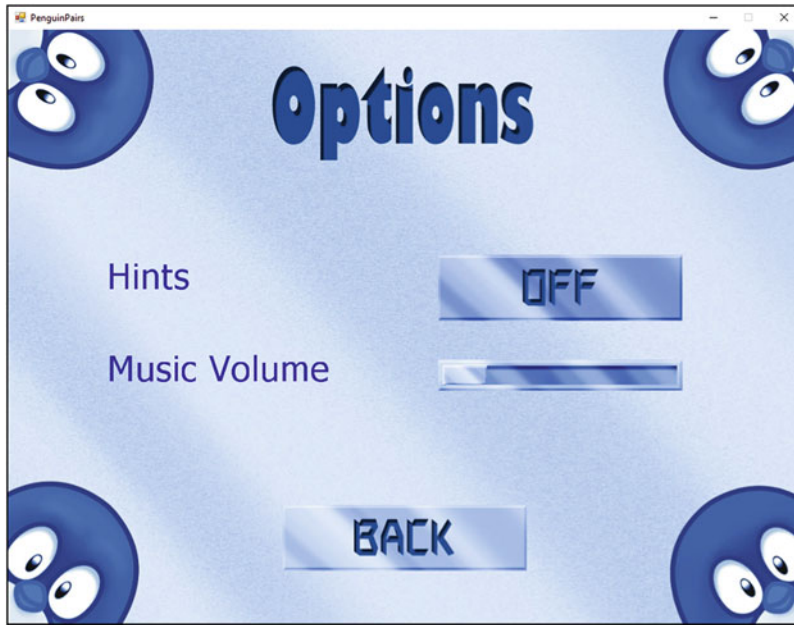


Fig. 18.3 The options menu of Penguin Pairs

for the button and slider, so that the `HandleInput` method can easily use them. For the two text objects, this isn't necessary, because you'll never have to refer to them in the rest of your code.

Give your `Slider` a minimum value of 0 and a maximum value of 1. (Later, we'll link this value to `MonoGame`'s music volume, and that's always a number between 0 and 1.) Give the slider a padding of 8: the background sprite for our slider has a border of 8 pixels thick.

For the exact *positions* of all UI elements, please take a look at the `PenguinPairs2b` example project. This project contains the full version of the `OptionsMenuState` constructor. You can also copy that code if you're unsure about your own version.

Once you've fully updated the `OptionsMenuState` constructor, compile and run the game again. All UI elements should now correctly respond to the player's actions.

Designing Menus and Options — Most games contain menu screens. With these screens, the player can change certain options, choose a different level, pause the game, and so on. Creating all of these additional screens can be a lot of work that seems unrelated to the actual gameplay. For that reason, developers tend to put less work into them—but that's a very dangerous choice.

Well-designed menus are a very important part of the player's overall experience, just like the gameplay itself. If a menu looks bad, or if it's unclear where to find certain options, then players may get frustrated by your game in general. Our advice: think carefully about what your menus look like, and let people test your menus before you publish the game. A user interface should be as clear and simple as possible!

Furthermore, think about how many options your game really needs. It may sound nice to make an option out of everything: the difficulty of the game, the music to play, the background

(continued)

color, and so on. But if you create too many options, you're almost letting players design the game themselves. It's usually better to determine most of these settings yourself (as a designer or programmer) and to only make the most important things changeable by the player.

18.4.2 Changing the Music Volume

The final step is to link the UI elements in the options menu to actual game settings.

The music volume is actually quite easy to change in MonoGame. The `MediaPlayer` class (part of the `Microsoft.Xna.Framework.Media` library) contains a static property `Volume`. This is a number between 0 and 1 that indicates the current music volume. If you change this value, the volume will change immediately. So, we can directly link the value of our slider (which also lies between 0 and 1) to that `Volume` property.

In the constructor of `OptionsMenuState`, right after you've created the slider, add the following line:

```
musicVolumeSlider.Value = MediaPlayer.Volume;
```

(assuming that `musicVolumeSlider` is the name of your member variable for the slider.) As you can probably guess, this instruction sets the initial value of the slider to the number that's currently stored as the music volume.

To update the volume when the slider's value changes, go to the `HandleInput` method of `OptionsMenuState` and add the following:

```
if (musicVolumeSlider.ValueChanged)
    MediaPlayer.Volume = musicVolumeSlider.Value;
```

In each frame, this code checks if something has changed in the slider; if so, the music volume gets updated accordingly.

That's all there is to it. Of course, you won't notice any effects until there's actually music in the game. We'll add background music in a later chapter—but you *could* already do it now, if you're eager to try out the slider.

18.4.3 Enabling and Disabling Hints

For the “hints” functionality, the easiest solution is to give the `PenguinPairs` class another static property:

```
public static bool HintsEnabled { get; set; }
```

(You could also use a *member variable* instead of a property. In this case, both solutions are pretty much equivalent.) We suggest setting this property to **true** in the constructor of `PenguinPairs`, so that hints are enabled by default. In the constructor of `OptionsMenuState`, you can use the current value of `HintsEnabled` to set the initial value of the “hints” switch:

```
hintsSwitch.Selected = PenguinPairs.HintsEnabled;
```

And in the `HandleInput` method, you can change the `HintsEnabled` property whenever the switch is pressed:

```
if (hintsSwitch.Pressed)
    PenguinPairs.HintsEnabled = hintsSwitch.Selected;
```

In the next chapter, you'll use this property to show or hide the "hint" button in `PlayingState` when a level starts.

More Game Settings — If you ever create a game with many more settings, it may be a good idea to store game settings in a more general way. Otherwise, the main class of your game will get flooded with static properties.

One option is to add a `Dictionary<string,string>` variable, where the keys of the dictionary are the *names* of your game settings. For example, you could use a key "playerName" to store the in-game name that the player has chosen. A disadvantage of this approach is that all settings in this dictionary would have to be strings. So, if you have a setting that represents a number or a Boolean value, you'll have to write extra code to convert between data types.

In the games of this book, there won't be any other game settings. Therefore, we choose to leave this topic for what it is.

18.5 Filling the Level Selection Screen

There's another menu that we need to fill in: the level selection screen. On this screen, we want the player to be able to select a level from a grid of level buttons. We won't add the actual levels yet in this chapter. For now, we'll just pretend that there are 12 levels.

18.5.1 Defining Level Statuses

A level button in the level selection screen can have three possible statuses: locked, unlocked, and solved. When the player first starts the game, only the first level will be unlocked, and all other levels will be locked. Locked levels cannot be played yet. Whenever the player finishes a level, that level's button will go to the "solved" status, and the next level will get unlocked. (We're using the word "status" here, because the word "state" is already in use for game states.)

Because the concept of a "level status" will be used by multiple classes later on, it's useful to define the three level statuses in an *enum*. Add a file `LevelStatus.cs` to your project, and fill it with the following simple code:

```
enum LevelStatus { Locked, Unlocked, Solved }
```

18.5.2 The *LevelButton* Class

To describe the behavior of a single button on the level selection screen, add a new class named `LevelButton`. A level button consists of two components: a button with a certain sprite and a piece

of text showing the number of the level. With the code you’ve prepared so far, you could define a `LevelButton` in at least two different ways:

1. A special type of `Button` that also draws text on top of itself.
2. A `GameObjectList` that contains a `Button` and a `TextGameObject` as its children.

If you go for option 1, you’ll have to give `LevelButton` extra code so that it draws the text as well. After all, a standard `Button` does not know how to draw text yet. If you go for option 2, you’ll have to give `LevelButton` extra code for checking if its `Button` child object has been pressed. Both options are okay, but in this book, we’ve chosen to go for option 1 because it results in slightly less code.

There’s actually a third option: you could choose to give the standard `Button` class an optional label. We haven’t done this in our code, but if you want to, you can do it yourself. If a regular `Button` already knows how to draw text, then `LevelButton` doesn’t have to do anything extra there, and inheriting from `Button` makes even more sense.

Our version of `LevelButton` is shown in Listing 18.5. Here’s an overview of what it does:

- It has extra member variables for storing the status of the associated level and for storing the `TextGameObject` of the label (so that it can call the `Draw` method for that object).
- For the *index* of the associated level, we’re using a property instead of a member variable, but with a private `set` component.
- The `Draw` method makes sure to draw the label on top of the button itself. We don’t have to override any other methods of the game loop.
- The constructor takes a level index and an initial level status as parameters and copies those into its member variables. It also initializes the `TextGameObject` to draw.
- The base constructor requires the name of a sprite to load. This should be the sprite for the button’s initial status. To be able to find that sprite, we’ve added a helper method `getSpriteNameForStatus` that returns the correct sprite name for a given level status.
- `getSpriteNameForStatus` chooses between three relevant sprites from the “Sprites/ UI” folder: *spr_Level_Locked*, *spr_Level_Unsolved*, and *spr_Level_Solved@6*. A level button should choose between these images depending on the status of the associated level. Note that the image for the “solved” status is a sprite sheet with six sprites, as indicated by the “@6” part of its name.
- When the player finishes a level, we expect that other game objects will want to change the status of a `LevelButton`. When that happens, the sprite of that button should change as well. To make this possible, we’ve added a `Status` property. In the `set` part of this property, the button does not only change the status member variable, but it also loads a new sprite, and it sets the sprite sheet index so that it depends on the level index. This makes sure that the buttons for solved levels don’t all look exactly the same.

All other behavior that we need is already part of the `Button` class, so we don’t have to program it again. Go ahead and add this `LevelButton` class to your game. You can find the file in the `PenguinPairs2b` example project. Feel free to make changes to the code if you have other preferences.

Listing 18.5 The `LevelButton` class

```

1 using Microsoft.Xna.Framework;
2 using Microsoft.Xna.Framework.Graphics;
3
4 class LevelButton : Button
5 {
6     /// <summary>
7     /// Gets the index of the level that this button represents.
8     /// </summary>
9     public int LevelIndex { get; private set; }
10

```

```

11 LevelStatus status;
12 TextGameObject label;
13
14 public LevelButton(int levelIndex, LevelStatus startStatus)
15 : base(getSpriteNameForStatus(startStatus))
16 {
17     LevelIndex = levelIndex;
18     Status = startStatus;
19
20     // add a label that shows the level index
21     label = new TextGameObject("Fonts/ScoreFont",
22         Color.Black, TextGameObject.Alignment.Center);
23     label.LocalPosition = sprite.Center + new Vector2(0,12);
24     label.Parent = this;
25     label.Text = levelIndex.ToString();
26 }
27
28 public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
29 {
30     base.Draw(gameTime, spriteBatch);
31     label.Draw(gameTime, spriteBatch);
32 }
33
34 /// <summary>
35 /// Gets or sets the status of this level button.
36 /// When you change the status, the button will receive a different sprite.
37 /// </summary>
38 public LevelStatus Status
39 {
40     get { return status; }
41     set
42     {
43         status = value;
44         sprite = new SpriteSheet(getSpriteNameForStatus(status));
45         SheetIndex = (LevelIndex - 1) % sprite.NumberOfSheetElements;
46     }
47 }
48
49 static string getSpriteNameForStatus(LevelStatus status)
50 {
51     if (status == LevelStatus.Locked)
52         return "Sprites/UI/spr_Level_Locked";
53     if (status == LevelStatus.Unlocked)
54         return "Sprites/UI/spr_Level_unsolved";
55     return "Sprites/UI/spr_Level_solved@6";
56 }
57 }

```

18.5.3 Adding a Grid of Level Buttons

You now have a class that can represent a level button. Let's use it in the `LevelMenuState` class to add a number of buttons to the level selection screen. You'll add these buttons as child objects. But because we need to know when a certain level button has been pressed, it's useful to give `LevelMenuState` another member variable:

```
LevelButton[] levelButtons;
```

Just like the `backButton` variable, the `levelButtons` variable will store extra references to the level buttons, so that we can easily find them back.

To actually add the level buttons, go to the end of the constructor method. As we've said before, we won't be loading any actual levels yet. Let's pretend that there are 12 levels, by adding the following code:

```
int numberOfLevels = 12;
levelButtons = new LevelButton[numberOfLevels];
```

The level buttons need to be placed in a grid, with five buttons per row. Start by defining a few parameters of this grid:

```
Vector2 gridOffset = new Vector2(155, 230);
const int buttonsPerRow = 5;
const int spaceBetweenColumns = 30;
const int spaceBetweenRows = 5;
```

After that, you can use a **for** loop to actually add the buttons:

```
for (int i = 0; i < numberOfLevels; i++)
{
    ...
}
```

In each iteration of that loop, you can start by creating a `LevelButton` object. We'll use `i+1` as the level index, to make sure that the first level will have a "1" as its label, and not a "0". Also, for now, let's give each button an initial status of `Solved`, just so that you can open all levels for testing purposes.

```
LevelButton levelButton = new LevelButton(i + 1, LevelStatus.Solved);
```

To give this button the correct position, you first need to determine the row and column in which the button is located. This information, together with the width and the height of a level button, can help you calculate the final position of the button:

```
int row = i / buttonsPerRow;
int column = i % buttonsPerRow;
levelButton.LocalPosition = gridOffset + new Vector2(
    column * (levelButton.Width + spaceBetweenColumns),
    row * (levelButton.Height + spaceBetweenRows)
);
```

After that, add the button to the game world, and store a reference in the `levelButtons` array:

```
gameObjects.AddChild(levelButton);
levelButtons[i] = levelButton;
```

This concludes the body of the **for** loop. If you compile and run the game now, there will be 12 level buttons in the level selection screen.

18.5.4 Starting a Level

As a final step, `LevelMenuState` needs to check if a level button has been pressed. If so, the game should start the corresponding level. Although we don't know how to load levels yet, you can already do

some of the work. Go to the `HandleInput` method of `LevelMenuState` and add the following code at the end:

```
foreach (LevelButton button in levelButtons)
{
    if (button.Pressed && button.Status != LevelStatus.Locked)
    {
        // go to the playing state
        ExtendedGame.GameStateManager.SwitchTo(PenguinPairs.StateName_Playing);

        // TODO: load the level
        return;
    }
}
```

This code uses a **foreach** loop to check all buttons in the `levelButtons` array. Because that's an array of `LevelButton` objects, you can immediately use the `Pressed` and `Status` properties in each iteration.

As soon as the program finds a level button that has been pressed and that doesn't belong to a locked level, it sends the game to the "playing" state. As a result, the level selection screen will then make place for the screen in which the player can actually play the chosen level. After changing the game state, we use a **return** keyword to make sure that the other buttons aren't checked anymore. This is not really required (because it's impossible to click multiple level buttons at once), but it saves the program some unnecessary work.



CHECKPOINT: PenguinPairs2b

Compile and run the code again. You can now go to the "playing" state by clicking a level button. You can press the Backspace key to go back.

This concludes the work for this chapter. In the next chapter, you'll load the actual levels from text files. This allows you to fill the menu with real level data, instead of the 12 "fake levels" we have now.

18.6 What You Have Learned

In this chapter, you have learned:

- how to handle sprite sheets in games;
- how to use the `Split` method to analyze strings;
- how to define different UI elements and add them to screens and menus;
- how to change the game when the player interacts with a UI element.

18.7 Exercises

1. *More UI Elements*

Can you think of other UI elements besides buttons, switches, and sliders? Name a few of them. Next, choose one of these UI elements and create a class for it. Test your class by adding an instance of it to the options menu.

2. *Pressing a Level Button*

In our example code, the `LevelMenuState` keeps checking if a level button has been pressed. You could also do this differently, by letting the level button itself check if it has been pressed, and (if so) sending the `LevelMenuState` a message that a level needs to be loaded.

How can you change the code to make this possible?

3. * *Better Buttons*

The `Button` class from this chapter is rather simple. For many applications, it's useful to have slightly "smarter" buttons that can respond to both *press* and *release* actions.

- a. Extend the `Button` class (e.g., by creating a subclass `ExtendedButton`) so that it does the following:
 - Instead of a single **bool** member variable, add an **enum** `ButtonState` with four states: `Normal`, `Hover`, `Pressed`, and `Released`. Give the `Button` class a member variable of that type. The default state should be `Normal`.
 - When the mouse pointer hovers over the button, the state should go to `Hover`.
 - When the player clicks the left mouse button while the mouse pointer is on the `Button`, the state should go to `Pressed`.
 - When the player releases the left mouse button, the `Button` class should check if the mouse pointer is still inside the button's bounding box. If so, the state should go to `Released`. If not, the state should go to `Normal`.
- b. It's common to show a different image for each button state. Allow your extended `Button` class to accept up to four different images, where the "normal" image is mandatory and the other images are optional. Change the class so that when the button's state changes, the sprite gets updated as well. Make sure that your code still works if some of the sprites are missing.
- c. In many games, you'll also want to *disable* a button temporarily. Extend the `Button` class with an `Enabled` property that other objects can get and set. When a button is not enabled, it should not respond to any mouse interaction, and it should (again) show a different sprite.

Chapter 19

Loading Levels from Files



Many games consist of different levels. In particular, puzzle games might even have *hundreds* of levels. With the programming tools you've seen so far, you could (for example) write a general `Level` class that describes the things that all levels have in common. You could then give `Level` hundreds of subclasses, one for each specific level in the game.

This approach has disadvantages: it leads to a lot of code, and you'll need to write a new class whenever you want to add another level to the game. In fact, this approach mixes the *game logic* (the rules and behavior of the game) with the *game content* (the layout of each specific level). In general, that's not a very nice way to program a game.

In this chapter, you'll work on a better solution. You will store the different levels in *text files*, where each file contains the layout of a specific level. You will add one single `Level` class that describes the game logic of *any* level. The content of each `Level` instance will depend on the specific file that you're loading.

The advantages of this approach should be obvious. If you now want to add another level to the game, you don't have to change the source code anymore. In a larger team of game developers, such an approach means that the programmers don't have to worry about creating levels anymore: they can leave that to the game designer, for example.

This chapter will teach you how to read and write text files in C#. This is sometimes also called *file I/O*, where the "I" stands for "input" and the "O" stands for "output."

At the end of this chapter, you'll also use file I/O to store the player's progress in a file, so that players don't have to re-unlock all levels each time they start the game. You will also learn about a new C# keyword (**switch**) that can sometimes be a handy alternative to **if** and **else** instructions.

19.1 Structure of a Level

If you want levels to be stored in text files, you need to determine what these files will look like. In a way, you'll design your own *file format* especially meant for your game. This file format depends on all the different elements that can be part of your levels.

In this section, we'll think about the structure of a Penguin Pairs level, and we'll design a file format around it.



Fig. 19.1 The first level of Penguin Pairs

19.1.1 All Possible Level Elements

Let's try to list the elements that can occur in a Penguin Pairs level. Fig. 19.1 shows a screenshot of the first level that we've prepared for you. We'll use that level as a recurring example in this section.

The level itself is always a 2D *grid*. When the player starts the level, each cell of this grid contains a certain type of element: a plain background block, a penguin of a certain color, a seal, a shark, a wall, a hole, or nothing at all. Penguins and seals could also already be *inside* a hole when the level starts.

Next to the grid, we want to give each level a short *title* and a longer *description* that explains the purpose of the level. We'll also give each level a *hint*: a suggestion for the first move that the player can do. A hint consists of a *grid position* and a *direction*. In the game, it will be shown as an arrow with that position and direction.

Finally, a level should define what the player needs to do to *solve* that level. You could say that a level is finished as soon as all penguins have disappeared from the grid. But this would mean that the player can also beat the game by killing all penguins, which isn't really the point of the game.¹ Instead, we'll say that a level is solved when the player has created a certain *number of penguin pairs*. This number can be different in each level.

¹Let's save that for the sequel.

19.1.2 Defining a File Format

Now that we know what the ingredients of a level are, we can decide how to place these ingredients in a text file. There are countless different ways to do this, and there's not really a "best" option. In this book, it's easiest if you follow our lead.

A level always has the same general information: a title, a description, a number of pairs that the player should make, and a hint arrow. Let's put each of these elements on its own line in the text file, in that order. Here are the first lines of our first example level:

```
Pair the Penguins
Click on a penguin. Then use the arrows to let it move.
1
4 3 down
```

The fourth line represents the hint arrow, consisting of a grid coordinate and a direction. We'll simply use the words "up," "down," "left," and "right" to indicate the direction. So, this example indicates an arrow on grid position (4, 3) that points down. We'll say that grid position (0, 0) refers to the top-left corner of the level, so there's a one-on-one correspondence with indices in the 2D array.

After this general information, the file should describe the level's grid. To make the file understandable for human readers, we'll use one text character per grid cell, and we'll put each row of the grid on a separate line. That way, the text will already look a lot like the game level that it represents. This also makes it easier to design new levels by hand.

If we want to fill a level based on a text file, it's important that our text files use *different characters* for each type of grid cell that we can have. Let's use the following rules:

- “#” indicates a wall, such as around the edges of the first level;
- “.” indicates a plain background block on which penguins and seals can stand;
- “ ” (an empty space) indicates an empty tile;
- any letter of “brgycp” indicates a penguin of a certain color (blue, red, green, yellow, cyan, or purple);
- “m” indicates a multicolored penguin;
- “x” indicates a seal;
- “_” indicates a hole in which penguins and seals can fall;
- any letter of “BRGYCPMX” (the capital versions of the letters we already defined) indicates a penguin or seal *inside* a hole;
- “@” indicates a shark.

For example, the grid of the first example level looks like this:

```
#####
#.....#
#...r...#
#.....#
#.....#
#.....#
#...r...#
#.....#
#####
```

Verify for yourself that this text matches the screenshot of Fig. [19.1](#).

We don't know in advance how large a level is; for example, we don't know how many rows there will be. But that doesn't matter: when reading a level file, we can just keep reading lines until we've

reached the end of the file. So, a level file doesn't have to say *explicitly* how large the grid is. We can simply find this out when the file gets loaded.

A grid is always as wide as its longest row in the level file. If a row happens to be shorter, we'll assume that the rest of that row is filled with empty tiles.

In the example assets for Penguin Pairs, you can find a "Levels" folder with 12 level files. The first level (our example so far) can be found in the file *level1.txt*. Have a look at the other level files as well, and recognize the elements we've described here.

19.2 Creating the Classes of Level Elements

For each symbol in (the grid part of) a level file that we're going to load, we'll have to add different game objects to the game. In this section, you'll create *classes* for the different objects that can appear in a level. It's useful to distinguish between two things:

- The objects in the grid that can move or disappear: penguins, seals, and sharks. We'll refer to these "dynamic" objects as *animals*.
- The parts of the grid that don't change. Each grid cell always has a certain "terrain type," such as a wall or an empty square. The type of a grid cell never changes during the game. We'll refer to these "fixed" objects as *tiles*.

In the end, a level will contain a 2D grid of tiles, plus a collection of animals that can move over this grid. In this section, you'll add classes for exactly these objects. To keep your code organized, it's a good idea to place these new classes in a new folder named "LevelObjects." Take a look at the PenguinPairs3a example project to see how we've done this.

Now's also a good time to include the sprites that these objects will need. From the example assets, include the entire "Sprites/LevelObjects" folder into your project. Also include the *spr_level_info* image (from the regular "Sprites" folder): this will be a background image for the level's general information.

19.2.1 The Tile Class

First, let's add a class that can store a *tile*. As explained before, a tile should represent the fixed "terrain type" of a grid cell. There are four possible tile types:

- a normal ice block on which penguins and seals can move;
- an empty square, representing the sea in which penguins and seals disappear;
- a wall that blocks penguins and seals;
- a hole in which penguins and seals can get stuck.

For "normal," "wall," and "hole" tiles, we want to draw a certain sprite. For "empty" tiles, we don't want to draw anything.

How does this translate to a `Tile` class? Well, a `Tile` is essentially a game object with a terrain type, plus an optional `SpriteGameObject` that it wants to draw. Also, it's convenient to create an **enum** with the four possible terrain types.

Listing 19.1 The `Tile` class

```

1 using Microsoft.Xna.Framework;
2 using Microsoft.Xna.Framework.Graphics;
3
4 class Tile : GameObject
5 {
6     public enum Type { Normal, Empty, Wall, Hole };
7
8     public Type TileType { get; private set; }
9
10    SpriteGameObject image;
11
12    public Tile(Type type, int x, int y)
13    {
14        TileType = type;
15
16        // add an image depending on the type
17        if (type == Type.Wall)
18            image = new SpriteGameObject("Sprites/LevelObjects/spr_wall");
19        else if (type == Type.Hole)
20            image = new SpriteGameObject("Sprites/LevelObjects/spr_hole");
21        else if (type == Type.Normal)
22            image = new SpriteGameObject("Sprites/LevelObjects/spr_field@2", (x + y) % 2);
23
24        // if there is an image, make it a child of this object
25        if (image != null)
26            image.Parent = this;
27    }
28
29    public override void Draw(GameTime gameTime, SpriteBatch spriteBatch)
30    {
31        // draw the image if it exists
32        if (image != null)
33            image.Draw(gameTime, spriteBatch);
34    }
35 }
```

Listing 19.1 shows our version of the `Tile` class. Go ahead and add it to your game. The constructor of `Tile` requires the type and coordinates of the tile to create. If the tile type is not `Empty`, the constructor loads the correct image. This image will have the `Tile` as its parent object, so that its global position will depend on the tile's position. The `Draw` method makes sure to draw the tile's image, if it exists.

If the tile type is `Normal`, the sheet index of the `SpriteGameObject` will be either 0 or 1, depending on the coordinates of the tile. (This is actually the only reason why `int x` and `int y` need to be passed to the constructor of a `Tile`.) The `spr_field@2` image is a sheet with two sprites of slightly different colors. By setting the sheet index to $(x + y) \% 2$, the tiles will form a nice “chessboard” pattern on the screen.

To let the last part of the code work, you have to give the `SpriteGameObject` constructor a second parameter (`int sheetIndex`) for the sheet index of its sprite. Add that parameter now, with a default value of 0. Also, make sure to pass this parameter to the `SpriteSheet` instance that `SpriteGameObject` creates.

19.2.2 The *Animal* Class

Next, let's add classes for the *animals* in the grid. All animals are special versions of a `SpriteGameObject`: they are objects with an image and a position in the game world. If you think about it, animals come in two flavors: *movable* animals that the player can control (penguins and seals) and *non-movable* animals (sharks).

This means that it's useful to create an `Animal` class first, just to indicate that the concept of an animal exists. This class can be abstract, because we'll never explicitly create an `Animal` instance: we'll always use a *subclass* of `Animal` instead. In short, `Animal` should be a simple abstract class that inherits from `SpriteGameObject`:

```
abstract class Animal : SpriteGameObject
{
    protected Animal(string spriteName, int sheetIndex = 0)
        : base(spriteName, sheetIndex) { }
}
```

This is all that the `Animal` class needs for now.

19.2.3 Subclasses of the *Animal* Class

Next, we'll add classes for the more specific types of animals: sharks, penguins, and seals. A shark is an animal that cannot move. Let's create a `Shark` class for this. For now, this class doesn't really do anything special, other than choosing the correct sprite to load:

```
class Shark : Animal
{
    public Shark() : base("Sprites/LevelObjects/spr_shark") {}
}
```

For the penguins and seals, you *could* create separate `Penguin` and `Seal` classes. However, these animals behave so similarly in `Penguin Pairs` that we've chosen to create one class for both: `MovableAnimal`. In fact, we're even going to use sprite sheets that combine the sprites for penguins and seals. The image `spr_penguin@8` contains seven penguin sprites and one seal sprite. The image `spr_penguin_holed@8` is an alternative version for when the animals are stuck inside a hole.

The `MovableAnimal` class is shown in Listing 19.2. It's a subclass of `Animal` with the following extra features:

- It has a member variable **bool** `isInHole` that says whether the animal is currently stuck inside a hole.
- It has an "animal index" between 0 and 7, representing the type of animal. This number is also the sheet index of the animal's sprite sheet. Therefore, you don't have to store it in an extra member variable: you can create a read-only property that returns `SheetIndex`.
- The constructor of `Animal` has two parameters: the animal index to use and the initial value for `isInHole`. The second parameter is useful because an animal can be stuck in a hole right from the start.
- An animal can fall into a hole during the game. When this happens, it should start using a different sprite sheet. To make this easier, we've added a property `IsInHole` that chooses a new sprite sheet in its **set** part.
- Because there are two places where the code needs to choose the correct sprite name to use, we've added a helper method for this: `GetSpriteName`. This method needs to **static**, because otherwise

we’re not allowed to use it in our call to the **base** constructor. This is because a “current instance” doesn’t exist yet at that point in the code: the program is still in the process of creating it!

We’ve made the `IsInHole` property public, but with a private **set** part. This means that all objects can *ask* if a `MovableAnimal` is inside a hole, but they can’t *change* this property. In the next chapter, you’ll let the animal itself check whether it has fallen into a hole.

Listing 19.2 The first version of the `MovableAnimal` class

```

1 class MovableAnimal : Animal
2 {
3     bool isInHole;
4
5     public MovableAnimal(int animalIndex, bool isInHole)
6         : base(GetSpriteName(isInHole), animalIndex)
7     {
8         this.isInHole = isInHole;
9     }
10
11     public int AnimalIndex { get { return SheetIndex; } }
12
13     static string GetSpriteName(bool isInHole)
14     {
15         if (isInHole)
16             return "Sprites/LevelObjects/spr_penguin_boxed@8";
17         return "Sprites/LevelObjects/spr_penguin@8";
18     }
19
20     public bool IsInHole
21     {
22         get { return isInHole; }
23         private set
24         {
25             isInHole = value;
26             sprite = new SpriteSheet(GetSpriteName(isInHole), AnimalIndex);
27         }
28     }
29 }

```

In the next chapter, you’ll add *behavior* to these animal classes. For now, it’s enough that the classes have a constructor and some basic data. This at least allows you to create the correct objects when you’ll load a level file.

But how does that loading actually work? How do you get information from a text file and use it in your game?

19.3 Reading and Writing Files

In this section, you’ll learn how to read and write text files in C#. Reading and writing files is also called **file I/O** (short for “file input and output”). We will focus on text files because they’re used in our example games, but we’ll also explain some more general concepts.

In the next section, you’ll use this knowledge to load actual level files. If you’re really eager to get started on that, you can jump to Sect. 19.4 more quickly. However, we strongly encourage you to first study *this* section carefully. It may be a bit tough, but it will really help you understand what’s going on when you read or write a file.

19.3.1 Introduction to File I/O

Every programming language handles file I/O a little bit differently. In that sense, file I/O is less “standard” than (for example) an **if** instruction or a **for** loop, which works exactly the same in many languages. However, the overall *concept* of reading and writing files is always similar. We’ll first explain the general ideas behind file I/O, so that you can easily teach yourself the details in other languages as well.

When you want to read or write a file, you first have to open some sort of “connection” between your program and the file. This connection is often called a *stream*. Once you’ve set up such a stream, you can read the contents of the file or you can write your own data to the file. Usually, when you set up the stream, you’ll have to specify if you’re going to read, write, or do a mix of both.

A file doesn’t always have to contain human-readable text. For example, an image file contains information about the colors of each pixel in the image. In the end, each file just contains *data*, and this data is essentially a long list of bytes. (Remember: a byte is a sequence of 8 bits in a computer’s memory, and it’s the main building block of everything that a computer can store.)

So, in general, file I/O is all about reading and writing bytes. However, because *text files* are so common, many programming languages (including C#) have special classes for reading and writing text.

The Easy Way — These days, C# offers a super-short way to read a text file. There’s a `File` class with a static method `ReadAllText` that you can call as follows:

```
string t = File.ReadAllText("myFile.txt");
```

where *myFile.txt* should be replaced by the name of your text file. After this instruction, the variable `t` will store the entire contents of the file that you’ve just read. Likewise, there’s a method `WriteAllText` that you may want to look at. In fact, the entire `File` class is worth checking out, if you’re interested.

These methods actually do quite a few things in the background. The libraries available in C# have many of these shortcuts for things that are used often. However, if you ever want to do something slightly different than what the shortcut does, you’ll need a different approach. In that case, it’s useful to know how everything works behind the scenes.

In this section, we’ll show you the “behind the scenes” version of file I/O. It may feel a bit old-fashioned compared to this single line of code, but we still believe that it’s a useful concept to understand.

19.3.2 I/O in C#: Streams, Readers, and Writers

In C#, there are many different classes related to I/O. The good news is that you don’t immediately have to understand all of them. If you just want to read and write text files, you’ll only be using three classes: `FileStream`, `StreamReader`, and `StreamWriter`. We’ll tell you the full story now, to help you understand things better later on.

There are basically three kinds of I/O classes in C#: streams, readers, and writers.

A **stream** represents something that you can read from and write to. This “something” is often a file, but it could also be a location in memory or several other things that don’t matter right now for

this book. There are different subclasses of `Stream` for the different types of sources you can use. For example, the `FileStream` class is used for reading and writing files. If your code contains the following line:

```
FileStream myStream = new FileStream("myFile.txt");
```

you can use the `myStream` object to read and write things to the file *myFile.txt*.

A stream is a bit of a vague concept. Intuitively, you could see a stream as the *connection* to a source: the invisible “cable” that connects your program to a file, for instance. In the example above, `myStream` is the object that connects your program to a file named *myFile.txt*. This intuitive description is probably not 100% accurate, but it’s good enough.²

With a stream alone, you can read and write *bytes* only. The `Stream` class has methods such as `ReadByte` and `WriteByte`, but not much more than that. The type of stream only indicates *where* you’re reading from (or writing to), but it doesn’t say yet what *kind* of data you’re reading or writing. For example, `FileStream` says that you’re dealing with a file, but it doesn’t yet say whether that file contains text, binary data, or something else. To read and write specific kinds of data, you’ll need a *reader* or a *writer*.

Readers are classes for reading specific things. You’ll mostly be using the `StreamReader` class in this book: this class is meant for reading *text* from a stream. The name `StreamReader` is a bit confusing, because it doesn’t have the word “text” in it.³ `StreamReader` has methods such as `ReadLine` and `ReadToEnd`, which read some text from a stream and return a string as a result. You’ll learn more about these methods soon.

Next to `StreamReader`, there is also a `BinaryReader` class for reading binary data. This class has methods such as `ReadByte`, `ReadDouble`, and `ReadInt32`, which read a certain chunk of the data and convert it to a **byte**, **double**, or **int**. There’s also an `XMLReader` class that’s specialized in reading files with the XML file format. However, you won’t need these classes unless you want to read things other than plain text.

Likewise, *writers* are classes for writing specific things. The idea is similar as with readers. For instance, `StreamWriter` lets you write pieces of text, and `BinaryWriter` lets you write integers, doubles, and other primitive data types. Again, you’ll use `StreamWriter` most of the time, because you’re mostly going to deal with text.

To summarize:

- A stream is a sort of “connection” for reading and writing bytes from/to a source (such as a file).
- A reader is an object for reading something more specific (such as text) from a source (which is usually a stream).
- A writer is an object for writing something more specific to a source.

A `Stream` allows you to both read and write from the same source, as well as *overwrite* parts of the source. By contrast, a reader (or writer) only allows you to read (or write), so they’re a bit more limited. In exchange, readers and writers are easier to use, because they allow you to use plain text and other intuitive things.

Fig. 19.2 is a diagram showing a number of these classes (and how they inherit from each other). Most of these classes are in the `System.IO` namespace, so you’ll often have to write **using** `System.IO`; in your C# code if you want to read or write something.

²At least it’s better than the word “something.”

³We would have preferred the name “`TextStreamReader`” instead, although that’s a pain to pronounce.

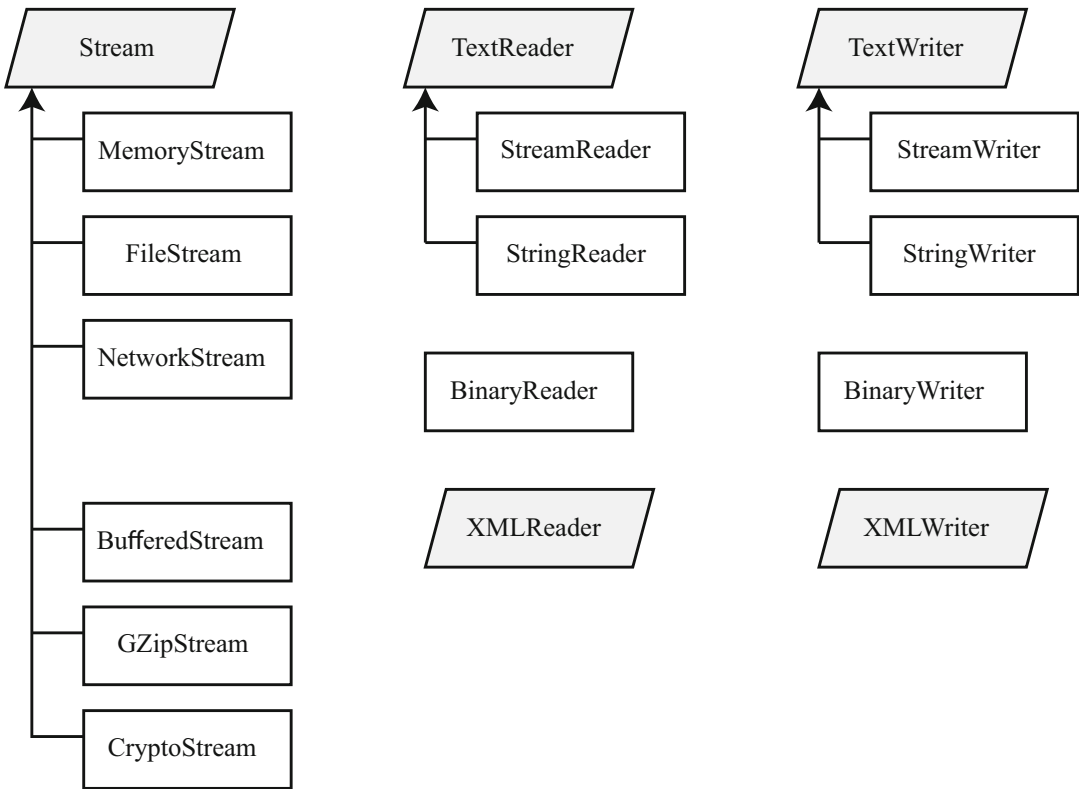


Fig. 19.2 The most common I/O classes in C#. A gray shape indicates an abstract class

Alternatives for Streams — We said that a stream is a “connection” to something you can read from and write to. Technically, it’s also possible to read and write things *without* requiring a stream. For example, there’s a `StringReader` class that lets you read text directly from another `String` object, without connecting to something external. `StringReader` and `StreamReader` both inherit from the (abstract) `TextReader` class, which deals with reading text in general. You can see this in Fig. 19.2 as well.

19.3.3 Reading a Text File in C#

The most common way to read a text file in C# is by using a `StreamReader` object. As we said earlier, a `StreamReader` is a “connection” that lets you read text from a stream. A `StreamReader` instance needs a certain kind of stream to read from. In the case of text files, we use an object of type `FileStream`.

To start reading a text file, you have to create the `FileStream` first and the `StreamReader` second:

```
FileStream myStream = new FileStream("myFile.txt");
StreamReader myReader = new StreamReader(myStream);
```

But because this combination of `FileStream` and `StreamReader` is so common, the authors of C# have created a special constructor for the `StreamReader` class that does both things at the same time. The following line:

```
StreamReader myReader = new StreamReader("myFile.txt");
```

is equivalent to the previous two lines combined. In the background, this constructor will create the `FileStream` object for you, so you don't have to worry about it anymore.

The `StreamReader` class has a number of useful methods for reading parts of the file. Here are the three most important ones:

- **char** `Read()` reads a single character from the file;
- **string** `ReadLine()` reads an entire line;
- **string** `ReadToEnd()` reads the entire file at once.

For example, given the `myReader` object, the following line:

```
string txt = myReader.ReadToEnd();
```

will read the entire file *myFile.txt* and store the result in the `txt` variable.

19.3.4 Reading One Line at a Time

You can use the methods `Read` or `ReadLine` to read only a certain part of the text file. As the `StreamReader` goes through the file, it keeps track of its current position in the file—as if it's moving a cursor through the text. Each time you call `Read` or `ReadLine`, this cursor will move one character or one line further.

So, if you call `ReadLine` repeatedly, that method will keep returning the next line of the file. When the cursor has reached the end of the file, `ReadLine` will return **null**. This means that you write a **while** loop such as this one:

```
string line = myReader.ReadLine();
while (line != null)
{
    Console.WriteLine(line);
    line = myReader.ReadLine();
}
```

This example will write the contents of the text file to the console, one line at a time. Of course, you can do other things with each line as well. You could put all lines in a `List<string>` and then process them later, like this:

```
List<string> lines = new List<string>();
string line = myReader.ReadLine();
while (line != null)
{
    lines.Add(line);
    line = myReader.ReadLine();
}
```

We'll use this approach in Sect. 19.4 as well, when reading the grid part of a level file.

When you've finished reading the file, it's good practice to call the `Close` method, which will close the connection to the file:

```
myReader.Close();
```

This instruction allows the program to clean up some memory, and it allows other programs to use the file as well. By the way, the `Close` method will automatically get called when the `myReader` object goes out of scope. For example, if `myReader` is a local variable inside a method, then the program will clean up the object when that method is finished. So, if you forget to call `Close` yourself, you'll often get away with it—but it's still good practice to write it yourself.



Quick Reference: File I/O (Reading)

To read a text file with the name *filename.txt*, first create a `StreamReader`:

```
StreamWriter r = new StreamReader("filename.txt");
```

You can then do various things:

- `r.Read()` will return a character in the file as a **char**;
- `r.ReadLine()` will return a line of the file as a **string**;
- `r.ReadToEnd()` will return the file's full text as a **string**.

Each time you call `Read` or `ReadLine`, you will receive the next character or line in the file. When you're done reading, you can close the file as follows:

```
r.Close();
```

19.3.5 Writing a Text File in C#

To *write* text to a file, you'll have to use the `StreamWriter` class instead of `StreamReader`. This class also needs to work with a stream, but it also has a special constructor that handles this for you in the background. For example, the following line:

```
StreamWriter myWriter = new StreamWriter("result.txt");
```

will open a connection to a new file named *result.txt*. Watch out: if that file already existed, it will be overwritten! If you want to open the existing file and add new text to it, you can use a different version of the constructor:

```
StreamWriter myWriter = new StreamWriter("result.txt", true);
```

The second parameter here indicates whether you want to add ("append") text to an existing file. If you pass **false** instead of **true**, you will overwrite the file again. (In other words, passing **false** is equivalent to passing no second parameter at all.)

Once the `StreamWriter` has been set up, you can use several methods to write data to the file:

- **void Write(string txt)** adds the string `txt` to the file.
- **void WriteLine(string txt)** adds the string `txt` to the file and then starts a new line.

There are also versions of these methods that write an **int**, **double**, and so on. Those methods will automatically convert your data to a string.

Just like `StreamReader`, a `StreamWriter` object keeps track of where it currently is in the file. This “cursor” is at the end of the text you’ve written so far. That way, the text will be added to the file in the same order as your instructions. `StreamWriter` also has a `Close` method, which works similarly as the one of `StreamReader`.

Here’s an example of a method that writes the numbers 1–1000 to a file, separated by spaces:

```
StreamWriter myWriter = new StreamWriter("result.txt");
for (int i=1; i<=1000; i++)
    myWriter.Write(i + " ");
myWriter.Close();
```

There’s really not much more to say about it. It’s better to just try it out with real data—so let’s start adding file I/O to Penguin Pairs!



Quick Reference: File I/O (Writing)

To write text to a file named *filename.txt*, first create a `StreamWriter`:

```
StreamWriter w = new StreamWriter("filename.txt");
```

If the file already existed, you will overwrite it. If you want to append an existing file instead, use a different constructor:

```
StreamWriter w = new StreamWriter("filename.txt", true);
```

You can use `w.Write(...)` to write a string to the file, or `w.WriteLine(...)` to write a string and then start a new line. When you’re done writing, close the file as follows:

```
w.Close();
```

19.4 Putting It to Practice: Reading a Level File

In this section, you’ll use a `StreamReader` to read the text of a level file. You’ll read the basic information of a level, followed by all lines of the level’s grid. Finally, you’ll convert those lines to an actual grid with `Tile` and `Animal` objects.

This section involves quite a lot of code, but most of it is very similar to what you’ve seen before. Of course, the most interesting part right now is related to file I/O. For the less interesting parts, we will sometimes leave out some code and refer to the `PenguinPairs3a` example project instead.

19.4.1 Outline of the *Level* Class

First of all, add a new `Level` class to the “LevelObjects” folder. This class will represent an entire level in the game: a grid with tiles and animals, plus some general level information. The `Level` class will

be somewhat comparable to the `JewelGrid` class from Jewel Jam, but with one important difference: a `Level` can fill its own grid by reading a text file.

`Level` should be a subclass of `GameObjectList`, because it's essentially a collection of game objects with their own behavior, plus some extra general information about the level. For this general information, give the class a property for the level's index and a private member variable for the number of penguin pairs required to complete this level:

```
public int LevelIndex { get; private set; }
int targetNumberOfPairs;
```

The title and description of a level *don't* need member variables of their own, because we'll never have to ask for them during the game. Instead, we'll just paste the title and description straight into a `TextGameObject` when we load the level.

As we've described before, the grid part of the level will be a collection of tiles, possibly with animals on them. These `Tile` and `Animal` objects will all be stored in the game-object list. However, during the game, we'll also want ask the level questions such as "what is the type of tile (2, 3)?" and "is there an animal at grid position (4, 1)?" To answer these questions quickly, it's useful to have 2D arrays that store extra references to the tiles and animals. Therefore, add the following two member variables:

```
Tile[ , ] tiles;
Animal[ , ] animalsOnTiles;
```

The level will also contain a hint arrow that should become visible when the player clicks on a button. Thus, it's useful to have an extra reference to that arrow:

```
SpriteGameObject hintArrow;
```

The constructor of a `Level` should have two parameters: a level index and the name of the file that contains the level's contents. It should store this level index internally and then load the level in some way. Let's assume that the file loading will be handled by a helper method named `LoadLevelFromFile`. The full constructor then looks like this:

```
public Level(int levelIndex, string filename)
{
    LevelIndex = levelIndex;
    LoadLevelFromFile(filename);
}
```

The rest of this section is all about writing the `LoadLevelFromFile` method.

There's one more thing you can already add right now: a method that converts grid coordinates to a position in the game world, just like we did for the `JewelGrid` class of Jewel Jam. All of our example sprites are 73×72 pixels large, so the width and height of a single grid cell are 73 and 72, respectively. Let's add two constants for that, at the top of the class:

```
const int TileWidth = 73;
const int TileHeight = 72;
```

Then give `Level` a method that calculates a `Vector2` out of two grid coordinates:

```
Vector2 GetCellPosition(int x, int y)
{
    return new Vector2(x * TileWidth, y * TileHeight);
}
```

You'll use this method a lot when adding the tiles and animals to the game world.

19.4.2 Reading the General Data

The `LoadLevelFromFile` method should start by creating a `StreamReader` that can read the level file. You can then read its first line to obtain the level's title, the second line to obtain the description, and the third line to obtain the required number of penguin pairs. For the third line, you need to use the `int.Parse` method to convert text to a number. So, the method starts out like this:

```
void LoadLevelFromFile(string filename)
{
    StreamReader reader = new StreamReader(filename);

    string title = reader.ReadLine();
    string description = reader.ReadLine();
    targetNumberOfPairs = int.Parse(reader.ReadLine());
    ...
}
```

Next, you can create game objects that show the level's title and description near the bottom of the screen. You'll need a `SpriteGameObject` that contains the `spr_Level_info` background image and two `TextGameObject` instances that contain the title and description. These objects should all be added to the game-object list (via the `AddChild` method).

In the `PenguinPairs3a` project, we've written a helper method (`AddLevelInfoObjects`) that adds these objects. Feel free to copy it into your class, or try to write the code yourself.

The fourth line of the file contains the hint arrow, written as two numbers and a word (such as "up" or "down") separated by spaces. First read the full line, and then use the `Split` method for strings to obtain the separate parts:

```
string[] hint = reader.ReadLine().Split(' ');
```

The resulting array has three elements. The first two elements store the *x*- and *y*-coordinates of the hint arrow:

```
int hintX = int.Parse(hint[0]);
int hintY = int.Parse(hint[1]);
```

To process the third element, you'll have to convert a word such as "up" or "down" into an arrow to draw. The example image `LevelObjects/spr_arrow_hint@4` is a sprite sheet with four arrow sprites, in the following order: right, up, left, and down. So, if we want to use this sprite to represent the four possible arrows, we need to convert the word "right" to a sheet index of 0, the word "up" to a sheet index of 1, and so on. Go ahead and write a helper method (`StringToDirection`) that converts a string to the correct number. You can also look at the `PenguinPairs3a` project for the answer.

Back in `LoadLevelFromFile`, you can now initialize `hintArrow` as a `SpriteGameObject` with the correct sprite and position. Make sure to use `StringToDirection` to find the correct sheet index and `GetCellPosition` to give the arrow the proper position in the game world.

Watch out: *do not* add the hint arrow to the game world yet! We haven't added the tiles and animals yet, and (eventually) the objects will be drawn in the order in which you've added them to the game world. So if you'd add the hint arrow now, you wouldn't be able to see it later, because the grid will be drawn on top of it. Instead, we'll add this hint arrow to the game world later.

19.4.3 Reading the Lines of the Grid

The rest of the text file contains the level’s grid. Because we don’t know how large the level will be, we should just keep reading the next line until we’ve reached the end of the file. The following code reads all lines and places them in a list of strings:

```
List<string> gridRows = new List<string>();
string line = reader.ReadLine();
while (line != null)
{
    gridRows.Add(line);
    line = reader.ReadLine();
}
reader.Close();
```

Note that we can now close the file, because we don’t have to do any more reading. The “file I/O” part of the work is now finished; the rest of this section is about translating each row to the correct game objects.

To convert these lines to actual game objects (tiles and animals), let’s add another helper method: `AddPlayingField`. This method (with return type **void**) should have three parameters: the list of strings you’ve read, the *width* of the grid to create, and the *height* of the grid. Add the header of this method now. We’ll add the body soon.

At the end of `LoadLevelFromFile`, we need to call this new method. But how can you determine the width and height of the grid? Well, the *height* of the grid is simply the number of lines that you’ve read: `gridRows.Count`. The *width* of the grid is a bit trickier to find, because each row in the file might have a different length. We said before that we’ll use the *longest* line to determine the overall width of the grid. So, we now have to find the longest string among the elements of `gridRows`. Try to add some code yourself that finds this longest row and stores its length in a local variable **int** `gridWidth`. You could use a **foreach** loop for this, or you could integrate it into the **while** that already exists.

After you’ve calculated the width, you can call your new method as follows:

```
AddPlayingField(gridRows, gridWidth, gridRows.Count);
```

Take a look at the `PenguinPairs3a` project to see if your code matches ours so far. Don’t worry about the contents of `AddPlayingField` yet.

19.4.4 Preparing the Grid

Let’s start filling in the `AddPlayingField` method. This method should convert the text of a level file (which we’ve just loaded) to a grid filled with tiles and animals. These tiles and animals will be instances of the `Tile` and `Animal` classes you’ve prepared in Sect. 19.1.

Again, we’ll skip over some of the “boring” code that adds game objects to the game world, because you’ve seen this many times already.

For convenience, the `AddPlayingField` method should start by creating a `GameObjectList` that contains all objects of the grid:

```
GameObjectList playingField = new GameObjectList();
```

Just like in Jewel Jam, this allows us to give the entire grid a certain position in the game world. In this case, we'd like this object to be roughly centered in the screen. So, the `LocalPosition` of the grid should depend on the *size* of the grid. Luckily, this size is easy to calculate. The following code places the center of the grid at position (600, 420), which leaves just enough room for the level's title and description at the bottom.

```
Vector2 gridSize = new Vector2(gridWidth * TileWidth, gridHeight * TileHeight);
playingField.LocalPosition = new Vector2(600, 420) - gridSize / 2.0f;
```

Next, initialize the two arrays of tiles and animals:

```
tiles = new Tile[gridWidth, gridHeight];
animalsOnTiles = new Animal[gridWidth, gridHeight];
```

Each time we find a tile or animal in our text file, we'll store it in this array.

19.4.5 Filling the Grid with Tiles and Animals

Now we need to extract the actual tile information from the list of strings (the `gridRows` parameter). Start with a nested **for** loop that iterates over all characters in all rows:

```
for (int y = 0; y < gridHeight; y++)
{
    string row = gridRows[y];
    for (int x = 0; x < gridWidth; x++)
    {
        ...
    }
}
```

First of all, remember that some rows may be shorter than others. We want to pretend that shorter rows are filled up with empty spaces. So, start the inner loop as follows:

```
char symbol = ' ';
if (x < row.Length)
    symbol = row[x];
```

This will set the `symbol` variable to the tile symbol that we're interested in. The default value is an empty space, unless the current row is long enough to say otherwise.

Next, we should create a tile and (possibly) an animal, based on the value of the `symbol` variable. There are many different cases here. For example, a `"#"` symbol represents a wall tile, a `":"` symbol represents a normal tile with no animal on it, and an `"r"` symbol represents a normal tile with a red penguin on it. Feel free to look at Sect. 19.1 again if you've forgotten the details of our file format.

If you think about it, each symbol *always* represents a tile, and it *sometimes* represents an animal as well. So, let's assume that we have a helper method `AddTile` that adds a `Tile` object to the `tiles` array and another method `AddAnimal` that *maybe* adds an `Animal` object to the `animalsOnTiles` array. We'll write those methods soon. Inside the inner **for** loop, you can then continue as follows:

```
AddTile(x, y, symbol);
AddAnimal(x, y, symbol);
```

This concludes the nested **for** loops. Assuming that the `AddTile` and `AddAnimal` methods work correctly, the `tiles` and `animalsOnTiles` arrays will be filled when the loops have finished. Each element of `tiles` will then contain a `Tile` object. Each element of `animalsOnTiles` will contain an `Animal` object or **null** if there is no animal on that tile.

After this nested loop, you still have to add the loaded objects to the game world. Because objects will be drawn in the same order in which you add them, it's important to first add the tiles and then the animals. So, start with a nested loop that adds all `Tile` objects as children to the `playingField` object:

```
for (int y = 0; y < gridHeight; y++)
    for (int x = 0; x < gridWidth; x++)
        playingField.AddChild(tiles[x, y]);
```

And after that, write a similar nested loop that adds all `Animal` objects as children of `playingField`. Go ahead and write this code yourself. Of course, you should take into account that some tiles do not have an animal. In other words, you should skip the grid cells where `animalsOnTiles` stores **null**.

Now that the grid has been filled, the final object to add is the hint arrow. Add that object to the game world now, and make sure that it's initially invisible:

```
hintArrow.Visible = false;
playingField.AddChild(hintArrow);
```

and then the `playingField` object is complete, so you can add it to the overall level:

```
AddChild(playingField);
```

This concludes the `AddPlayingField` method. But of course, we still need to write the `AddTile` and `AddAnimal` methods.

19.4.6 The *AddTile* Method

The `AddTile` method should create a `Tile` instance with a certain type, using the `Tile` constructor that you've already prepared a few sections ago. Next, it should give that tile the correct position in the game world, and it should store the object inside the `tiles` array.

Once again, it's useful to create a helper method here. This time, we'll add a method `CharToTileType` that converts a **char** (a symbol from our text file) to the correct `Tile.Type` (the tile type to pass to the `Tile` constructor). Assuming that this method exists, the `AddTile` method should look like this:

```
void AddTile(int x, int y, char symbol)
{
    Tile tile = new Tile(CharToTileType(symbol), x, y);
    tile.LocalPosition = GetCellPosition(x, y);
    tiles[x, y] = tile;
}
```

For the `CharToTileType` method, there are four symbols that are easy to handle. These correspond to tiles of a certain type, *without* an animal on them:

```
Tile.Type CharToTileType(char symbol)
{
    if (symbol == ' ')
        return Tile.Type.Empty;
    if (symbol == '.')
        return Tile.Type.Normal;
    if (symbol == '#')
```

```

    return Tile.Type.Wall;
if (symbol == '.')
    return Tile.Type.Hole;
...
}

```

In all other cases, we're dealing with a tile that has an animal on it. This can be a normal tile (if the symbol is an "@" or a lowercase letter of "brgycpmx") or a hole tile (if the symbol is an uppercase letter of "BRGYCPMX"). So, a quick and easy solution is to write the following:

```

if ("BRGYCPMX".Contains(symbol))
    return Tile.Type.Hole;
return Tile.Type.Normal;

```

This uses the `Contains` method from the `String` class to check if `symbol` is any of the "hole"-related letters. If so, then we're dealing with a hole tile; if not, then it's a normal tile.

19.4.7 The *AddAnimal* Method

The `AddAnimal` method should check if a symbol represents a tile with an animal on it. If so, it should create the correct `Animal` object and place it in the `animalsOnTiles` array. Start the method as follows:

```

void AddAnimal(int x, int y, char symbol)
{
    Animal result = null;
    ... // TODO: check if the symbol represents an animal

    if (result != null)
    {
        result.LocalPosition = GetCellPosition(x, y);
        animalsOnTiles[x, y] = result;
    }
}

```

At "...", you'll add the different cases that can occur. Remember that a tile could contain a shark, a movable animal, or a movable animal inside a hole. If any of these cases occurs, you should create a new `Animal` instance and store it in the `result` variable. At the end of the method, you'll then give that `Animal` the correct position and add it to the array.

What do these different cases look like? Well, a "@" symbol indicates a tile with shark, so in that case, we should create a `Shark` instance:

```

if (symbol == '@')
    result = new Shark();

```

Lowercase letters of "brgycpmx" indicate a movable animal that is not inside a hole. In those cases, you'll have to create a `MovableAnimal`, but you'll first have to determine the correct sprite sheet index to use. (Remember that the penguins and the seal share one sprite sheet with eight sprites: a blue penguin, a red penguin, and so on.)

Actually, this sheet index is exactly the position at which the symbol character appears in the sequence "brgycpmx". To find this position, you can use the `IndexOf` method of the `String` class. This method returns the index at which a certain `char` occurs in a string. If the character does not occur in the string at all, the method returns `-1`.

In short, the following code handles the “lowercase letters” scenario:

```
if (result == null)
{
    int animalIndex = "brgycpmx".IndexOf(symbol);
    if (animalIndex >= 0)
        result = new MovableAnimal(animalIndex, false);
}
```

This will create a `MovableAnimal` with the correct sprite sheet index. Remember that the second parameter of the `MovableAnimal` constructor indicates whether or not the animal is inside a hole; that’s why we’re using **false** here.

Also, note that this code is wrapped inside an **if** statement. This is because we don’t have to do anything if we’ve already determined earlier that this tile contains a shark.

The “uppercase letters” scenario (for animals inside a hole) works similarly. However, it uses a different string to check, and it passes a different Boolean value to the `MovableAnimal` constructor:

```
if (result == null)
{
    int animalIndex = "BRGYCPMX".IndexOf(symbol);
    if (animalIndex >= 0)
        result = new MovableAnimal(animalIndex, true);
}
```

And that’s it! The `AddAnimal` method is now finished—and with that, the constructor of `Level` is finished as well.

Cleaning Up the Code — You may notice that we’re using the string “BRGYCPMX” (and its lowercase brother) in multiple places now. In the minds of experienced programmers, this should sound a couple of alarms. If we ever want to *change* this string, we’ll have to remember to do it in several places!

In our `PenguinPairs3a` project, we’ve cleaned up the code a little bit. We’ve converted that string to a constant:

```
const string MovableAnimalLetters = "brgycpmx";
```

and we’ve created two helper methods that do the `IndexOf`-related work:

```
int GetAnimalIndex(char symbol)
{
    return MovableAnimalLetters.IndexOf(symbol);
}
int GetAnimalInHoleIndex(char symbol)
{
    return MovableAnimalLetters.ToUpper().IndexOf(symbol);
}
```

`ToUpper` is a nice helper method from the `String` class that turns all lowercase letters of a string into uppercase letters (capitals).

We’re using these two methods as much as possible, so that the sequence of letters is defined in only one place. Take a look at the example project if you’re curious.

(continued)

As you become a more experienced programmer, you'll (hopefully) develop a stronger urge to write clean and reusable code. In this book, we sometimes show you the "quick and dirty" option on purpose—simply because the "nicest" solution would take too many words to explain. However, we'll keep reminding you of the importance of code quality, for example, with gray boxes like this one.

19.5 Adding Levels to the Game

You've now written the code that can load a level when the game asks for it. However, you still need to add the level files to the project, *and* make sure that the player can load a specific level by choosing it in the level selection screen.

In this section, you'll implement exactly those steps. First, you'll add the text files to the project. Second, you'll put the `PlayingState` class in charge of showing one level at a time. Finally, you'll update the `LevelMenuState` class so that it loads a level when the player clicks on a level button. At the end of this section, you'll be able to see the levels on your screen.

19.5.1 Adding the Level Files to the Project

You could say that the text files of our levels are assets, just like sprites and sounds. For consistency, let's add the level files via the Pipeline Tool, so that all assets are wrapped inside the *Content.mgcb* object. In the Pipeline Tool, create a subfolder named "Levels," and add the 12 level files from our example assets (*level1.txt*, *level2.txt*, and so on).

There's one extra thing you need to do. Based on the file type, the Pipeline Tool will automatically try to determine what to do with an asset. For instance, it knows that *.jpg* and *.png* files should be converted to textures and that *.wav* and *.mp3* files are probably songs or sound effects. However, it doesn't know what to do with text files. In this case, we don't want MonoGame to convert these text files to anything special. We just want to have the "raw" text available when we need it.

To make this possible, click on a text file in the Pipeline Tool. In the "Properties" panel below, change the "Build Action" from *Build* to *Copy*, as shown in Fig. 19.3. This will prevent MonoGame from trying to convert the text file to something else. Instead, it will simply let MonoGame copy the text file to the location that will contain your game's final executable. You have to do this for all 12 level files. Sorry about that!

19.5.2 Storing a Level in the Playing State

The `PlayingState` class currently only shows a background image and some buttons. But now that we have a `Level` class, we can make this game state in charge of showing an actual level. Give `PlayingState` a new member variable:

```
Level level;
```

We can't initialize this variable in the constructor yet, because no level has been loaded when the game starts. We'll give `level` a value as soon as the player chooses a level in the level selection screen.

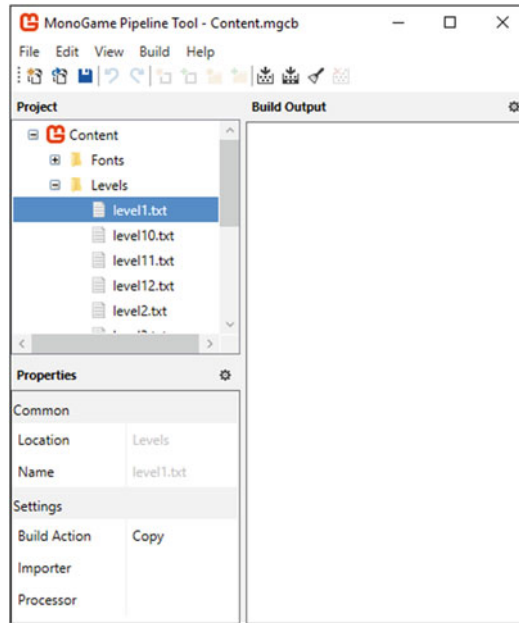


Fig. 19.3 The properties of the file *level1.txt* in the Pipeline Tool

When level does have a value, *PlayingState* should be in charge of showing and updating this level. In other words, it should call the *HandleInput*, *Update*, and *Draw* methods of *level* at the appropriate times. Therefore, add the following code to the *HandleInput* method:

```
if (level != null)
    level.HandleInput(inputHelper);
```

You'll have to call *level.Update* and *level.Draw* similarly. *PlayingState* doesn't override these methods yet, so please add them now. For example, *Update* should look like this:

```
public override void Update(GameTime gameTime)
{
    base.Update(gameTime);
    if (level != null)
        level.Update(gameTime);
}
```

The *Draw* method is very similar; go ahead and write it yourself.

19.5.3 Letting the *Playing State* Load a Level

Next, we'll add a method that loads a level and stores it in the *level* member variable. Basically, the *PlayingState* class should be able to load a level with a certain index, and other objects should be able to trigger that task. This logically leads to the following method header:

```
public void LoadLevel(int levelIndex)
```

Our example level files are conveniently named *level1.txt*, *level2.txt*, and so on. So, the filename for a level with index `levelIndex` is given by the following expression:

```
"Content/Levels/level" + levelIndex + ".txt"
```

You can immediately pass this filename to the constructor of the `Level` class. Add the following instruction to the `LoadLevel` method:

```
level = new Level(levelIndex, "Content/Levels/level" + levelIndex + ".txt");
```

This will load the correct level and store it in the `level` member variable.

By the way, the `LoadLevel` method is also a good place to make the “hint” button visible or invisible. That way, whenever a level gets loaded, the “hint” button will adapt its visibility to the most recent game settings. So, add the following instruction as well:

```
hintButton.Visible = PenguinPairs.HintsEnabled;
```

19.5.4 Loading a Level When the Player Asks for It

Almost there! The `LoadLevel` method is now finished, but you still need to *call* it at the right moment. This should happen in the `HandleInput` method of `LevelMenuState`. That method currently already checks if a level button has been pressed. If so, it switches the game to the “playing” state:

```
ExtendedGame.GameStateManager.SwitchTo(PenguinPairs.StateName.Playing);
```

If you’ve followed the book so far, there should be a “TODO” comment below it. You can now finally replace that comment by code that actually loads the level. First, retrieve the `PlayingState` object, like this:

```
PlayingState playingState = (PlayingState)ExtendedGame
    .GameStateManager.GetGameState(PenguinPairs.StateName.Playing);
```

and then load the level that matches the index of the clicked button:

```
playingState.LoadLevel(button.LevelIndex);
```

Casting to the `PlayingState` type is necessary here, because only the `PlayingState` class has a `LoadLevel` method. You can’t call the `LoadLevel` method if you only have an ordinary `GameState` object.



CHECKPOINT: PenguinPairs3a

Compile and run the game again. If you now select a level from the menu, you should see the corresponding level on your screen.

19.6 Reading and Writing the Player's Progress

We can also use file I/O to keep track of which levels the player has unlocked and solved. This allows the player to quit the game and continue later without losing their progress.

In this section, you’ll design a way to read this progress from a text file and to update this file when the player solves a level. You’ll also use this file to fill the level selection screen in a nicer way.

19.6.1 Representing the Player's Progress

To store the player's progress in the game, we basically need to store a `LevelStatus` value for each level that exists. Give the main `PenguinPairs` class the following member variable:

```
static List<LevelStatus> progress;
```

We'll fill this list when we load the player's progress from a text file. Also, let's give the `PenguinPairs` class a property that returns the length of this list, which indicates the total number of levels in the game:

```
public static int NumberOfLevels { get { return progress.Count; } }
```

Just like any other list, the progress member variable starts at index 0. However, the levels themselves will be numbered from 1 and onward. Therefore, `progress[0]` will store the status of level 1, `progress[1]` will store the status of level 2, and so on. Because this may be a bit confusing, let's give the `PenguinPairs` class two convenient methods for getting and setting the status of a certain level:

```
public static LevelStatus GetLevelStatus(int levelIndex)
{
    return progress[levelIndex - 1];
}
static void SetLevelStatus(int levelIndex, LevelStatus status)
{
    progress[levelIndex - 1] = status;
}
```

The `GetLevelStatus` method is public, because other classes (such as the level selection screen) will need it. The `SetLevelStatus` method is private, because the `PenguinPairs` class itself will be responsible for updating the player's progress.

19.6.2 Loading the Progress File

In our example assets, you can find a file *levels_status.txt* that contains the initial status of all levels. This file starts out as follows:

```
unlocked
locked
```

followed by ten more lines with the word "locked." The meaning of this file should be clear: each line corresponds to a level and says whether that level is locked, unlocked, or solved. Initially, only the first level is unlocked, and the rest is locked. Add this file to the "Levels" folder of your assets, just like how you've added the level files themselves. Don't forget to set the build action to "Copy" again.

To create and fill the progress member variable, give the `PenguinPairs` class a `LoadProgress` method. This method should initialize the progress variable as an empty list and then read the *levels_status.txt* file. For each line in the file, it should add an element to the progress list, depending on the text of that line. The strings "locked", "unlocked", and "solved" all correspond to a value of the `LevelStatus` enum.

Try to write this `LoadProgress` method yourself as an exercise. Have a look at the `PenguinPairs3b` project for the answer. You can keep the method private, because only the `PenguinPairs` class itself will be calling it.

Call this method when the game starts. The `LoadContent` method of `PenguinPairs` is a good place. Make sure to call `LoadProgress` *before* creating the different game states. That way, the game states can already use the `PenguinPairs.NumberOfLevels` property if they want to.

19.6.3 Saving a New Progress File

Similarly, you can give this class a `SaveProgress` method that writes the player's current progress to the same file. This method is quite simple: it sets up a `StreamWriter`, loops over the elements of the progress list, and writes a certain word for each element. Here's the full code:

```
static void SaveProgress()
{
    StreamWriter w = new StreamWriter("Content/Levels/levels.status.txt");
    foreach (LevelStatus status in progress)
    {
        if (status == LevelStatus.Locked)
            w.WriteLine("locked");
        else if (status == LevelStatus.Unlocked)
            w.WriteLine("unlocked");
        else
            w.WriteLine("solved");
    }
    w.Close();
}
```

When should this method get called? Eventually, we'd like to save the game each time the player finishes a level. To prepare for that, give the `PenguinPairs` class the following method:

```
public static void MarkLevelAsSolved(int levelIndex)
{
    SetLevelStatus(levelIndex, LevelStatus.Solved);

    if (levelIndex < NumberOfLevels && GetLevelStatus(levelIndex + 1) == LevelStatus.Locked)
        SetLevelStatus(levelIndex + 1, LevelStatus.Unlocked);

    SaveProgress();
}
```

This method first sets the status of the given level to `Solved`. Next, if this level is *not* the last level, and the next level has not yet been unlocked, this method marks the next level as `Unlocked`. Finally, the method calls `SaveProgress` to save the new level statuses to a text file.

In the final chapter of this part of the book, you'll call the `MarkLevelAsSolved` method when the player actually finishes a level.

19.6.4 Showing the Player's Progress in the Level Menu

The last step is to fill the level selection menu based on the player's progress. So far, we've pretended that there are 12 levels, and we've given all buttons the `Unlocked` status. But now that we're retrieving the player's progress information from a file, you can finally fill this screen with the truth.

To do this, you only need to make two small changes in the `LevelMenuState` class. Try to make these changes yourself:

- Instead of creating a “hard-coded” 12 buttons, create exactly as many buttons as there are levels in the game, using the `PenguinPairs.NumberOfLevels` property.
- Instead of giving all buttons a fixed status, give each button the status that is stored in the `PenguinPairs.Progress` list, using the `PenguinPairs.GetLevelStatus` method.

For now, it’s a good idea to update your `levels_status.txt` file so that all 12 levels are marked as unlocked. Because we haven’t yet written any code that lets players *finish* a level (or even play it in the first place), this is currently the only way to see all levels in action.



CHECKPOINT: PenguinPairs3b

Compile and run the game again. The level selection screen now gets filled based on the progress text file.

Congratulations: you’ve now completed the work for this chapter!

19.7 The **switch** Instruction: Handling Many Alternatives

In this chapter, you’ve written several pieces of code with the following structure:

```
if (variable == someExpression)
    // do something
else if (variable == someOtherExpression)
    // do something else
// ...and so on
```

Because this structure (with many **if** and **else** instructions in a row, checking the same variable) occurs a lot, C# offers a different way of writing it down: the **switch** instruction. It’s especially meant for when you’re checking a single variable against many different cases.

The **switch** instruction takes some getting used to, but it has advantages as well. We’ll use the Penguin Pairs code to explain how it works. Because **switch** is really just an alternative version of something you already knew, you don’t *have* to use it in your own project. It’s up to you to decide if you like the **switch** instruction enough to use it.

19.7.1 Basic Usage of **switch**

A **switch** instruction allows you to specify all the alternatives that can occur, along with the instructions that should be executed for each alternative. Let’s take the recent `SaveProgress` method as an example. It contains a **for** loop with the following body:

```
if (status == LevelStatus.Locked)
    w.WriteLine("locked");
else if (status == LevelStatus.Unlocked)
    w.WriteLine("unlocked");
else
    w.WriteLine("solved");
```

This can be rewritten to the following **switch** instruction:

```
switch (status)
{
    case LevelStatus.Locked:
        w.WriteLine("locked");
        break;
    case LevelStatus.Unlocked:
        w.WriteLine("unlocked");
        break;
    default:
        w.WriteLine("solved");
        break;
}
```

As you can see, the main advantage here is that you don't have to write the "status ==" part all the time. When the program executes a **switch** instruction, it will first calculate the value of the expression in parentheses behind the keyword **switch**. In this case, it will check the value of the status variable. Next, the program will look for the **case** keyword that has that exact value behind it. The program will then execute all instructions immediately after that, until it encounters the special **break** instruction.

You've seen the keyword **break** before in Chap. 9; it can also be used to jump out of a *loop*. It turns out that the keyword has a second meaning, specifically for a **switch** block. In C#, all cases in a **switch** instruction should end with a **break** instruction (or with a **return** instruction, but more on that later).

The keyword **default** is a special case: the code in this case will be executed if none of the other cases occur. The **default** case is comparable to the final **else** in a long sequence of **if** and **else** instructions. It's common to write your **default** case at the very end of the **switch** block, like we did in this example.

By the way, you don't *have* to write a default case. The default case is mandatory in some programming languages, but not in C#.

Sometimes, you'll have multiple cases in which you want to do the exact same thing. When using **if** instructions, you could write something like this:

```
if (variable == someExpression || variable == someOtherExpression)
    // do something
```

In a **switch** instruction, you can achieve the same effect by writing multiple cases directly below each other, followed by the shared code for all cases. For example, let's say we are checking a variable **char** symbol and we want to do a certain task when symbol stores the value "a," "b," or "c." You could then write the following:

```
switch (symbol)
{
    case 'a':
    case 'b':
    case 'c':
        // shared code for a, b, and c
        break;
    // ...other cases
}
```

19.7.2 Using *return* Instead of *break*

To get more familiar with the **switch** instruction, let's take the `CharToTileType` method of the `Level` class as another example. This method returns a certain `Tile.Type` value based on the **char** that it receives as a parameter. We can rewrite this to use the **switch** keyword instead of the many **if** instructions:

```
Tile.Type CharToTileType(char symbol)
{
    switch (symbol)
    {
        // standard cases
        case ' ': return Tile.Type.Empty;
        case '.': return Tile.Type.Normal;
        case '#': return Tile.Type.Wall;
        case '_': return Tile.Type.Hole;
        // every other symbol can be either a hole or a normal tile
        default:
            if (GetAnimalInHoleIndex(symbol) > 0)
                return Tile.Type.Hole;
            return Tile.Type.Normal;
    }
}
```

The main difference to the previous **switch** example is that we're not using the **break** keyword here. Instead, each case contains the **return** keyword. This is also allowed: the important thing is that the code of every case stops at some point. In a **switch** block, the **return** keyword does this just as well as the **break** keyword. However, **return** has the "side effect" that the program immediately jumps out of the current method. In this case, that's exactly what we want. In the `SaveProgress` example, writing **return** would have been wrong, though. Do you see why?

19.7.3 Restrictions

There are some special rules that you need to know about. A **switch** instruction only works if the expression between parentheses has a data type with a limited number of possible values, such as **int**, **char**, **bool**, **string**, or any enum type. Also, the expression behind each **case** keyword has to be a constant: the compiler should understand its value before the program starts.

So, the **switch** instruction is a bit less flexible than **if** and **else**. In exchange, a **switch** block can often be compiled into a more efficient program. Also, a **switch** block is sometimes easier to read, if the cases are simple enough.⁴



Quick Reference: The **switch** Instruction

A **switch** instruction is an alternative way of writing several **if** and **else** instructions in a row. Assuming that `x` is any integer expression, the following code with **if** and **else** instructions:

(continued)

⁴Not everybody agrees on that, though. The Internet is full of discussions about this topic, and even the authors of this book have clashing opinions. We still plan to settle this with some kind of battle.

```

if (x==1)          one();
else if (x==2)      { two(); alsoTwo(); }
else if (x==3 || x==4) threeOrFour();
else               somethingElse();

```

is equivalent to the following **switch** block:

```

switch (x)
{
    case 1: one();
            break;
    case 2: two();
            alsoTwo();
            break;
    case 3:
    case 4: threeOrFour();
            break;
    default: somethingElse();
            break;
}

```

Each case should end with a **break** instruction (or a **return** instruction, but then you'll also jump out of the current method). The **default** case is optional.

A **switch** instruction only works if the expression *x* has a simple data type (such as **int** or **char**), and if the expression behind each **case** keyword is a constant.

19.8 What You Have Learned

In this chapter, you have learned:

- how to read and write text files in C#;
- how to create a tile-based game world;
- how to retrieve and store the player's progress using a file;
- how a **switch** instruction can replace a sequence of **if** and **else** instructions.

19.9 Exercises

1. File Reading

Write a method `ReadSum` that takes a filename as its argument. You may assume that the file with that filename contains integers separated by spaces. Your method should read that file, calculate the sum of the numbers in the file, and return that sum.

For example, if the file *numbers.txt* contains the following text:

```
4 8 15 16 23 42
```

the method call `ReadSum("numbers.txt")` should return the number 108.

Hint: Split the string into separate numbers first, and then convert each part to an integer.

2. *File Writing*

Write a method `WriteMississippi` that takes two arguments: a filename and a positive integer. This method should create a new text file with the given filename and fill it with the text “1 Mississippi”, “2 Mississippi”, and so on, up to the given number. Each number should start on a new line.

For example, the method call `WriteMississippi("count.txt", 5)` should result in a file *count.txt* with the following text:

```
1 Mississippi
2 Mississippi
3 Mississippi
4 Mississippi
5 Mississippi
```

3. *Closing a Reader or Writer*

The `Close` method of a `StreamReader` or `StreamWriter` object will automatically be called when the object goes out of scope. Why is it still a good idea to call the `Close` method yourself?

4. *Making the **switch***

In the code of *Penguin Pairs* so far, are there any other places where a **switch** instruction would make sense? Try to find at least one such code fragment and replace its **if** and **else** instructions by a **switch** block.

Chapter 20

Gameplay Programming



You're now at the point where you can load the different Penguin Pairs levels by selecting them from a menu. In this chapter, you'll program the main *gameplay* of Penguin Pairs. We'll discuss how to select animals (penguins and seals) on the board, how to let them move around, and what to do when game objects collide.

You can program this part of the game in many different ways, and our way is just one of them. In our version, we choose to keep the *game logic* separated from the *visualization* as much as possible. For example, when we say that two objects "collide," we mean (in this game) that two objects have reached the same grid cell or two neighboring grid cells. Unlike in the Painter game, we won't actually check if there is a "physical" collision between the bounding boxes of objects. Although our approach might be a bit more difficult to program, we believe it leads to better code.

This chapter will teach you one new C# programming concept: the keyword "**is**". Mostly, though, the chapter is another exercise in writing complicated game logic, using the programming concepts you already know.

At the end of this chapter, the game will be almost entirely playable!

20.1 Selecting Animals

Before the player can move penguins and seals around, he/she needs to be able to *select* an animal. When the player clicks on an animal, we'd like to show four arrows around the animal, indicating the directions in which the animal can move. See Fig. 20.1 for an example. Then, when the player clicks on one of these arrows, the animal should start moving in that direction (if possible).

In this section, you'll add this selection functionality to the game. You'll add two classes: *Arrow* (representing a single arrow) and *MovableAnimalSelector* (the parent object that contains all four arrows).

20.1.1 The *Arrow* Class

An arrow is essentially a button on which players can click. Let's add a nice visual effect, though: when the player moves the mouse pointer over one of the arrows, the arrow will become a bit darker.

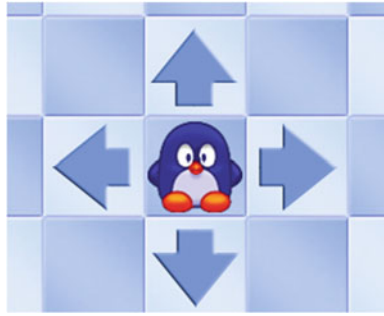


Fig. 20.1 When the player clicks on a penguin, four clickable arrows should appear



Fig. 20.2 A sprite sheet containing four arrows

This means that an arrow should be a button with an extra feature: it should show a different sprite when the mouse pointer hovers over it.

For the sprite of an arrow, we’ve provided a sprite sheet that contains four different arrows pointing in all four directions: see Fig. 20.2. We’ve also provided a “hover” version of this sprite sheet. Add those two sprite sheets to the project now.

Create a class `Arrow` (in the “LevelObjects” folder) that inherits from `Button`. Make sure that its constructor has a parameter indicating the sprite sheet index, because that’s what determines the direction of the arrow.

Listing 20.1 shows our full version of the `Arrow` class. As you can see, the class has two `SpriteSheet` member variables, and it switches between them depending on whether the mouse is hovering above the arrow.

Listing 20.1 The `Arrow` class

```

1 class Arrow : Button
2 {
3     SpriteSheet normalSprite, hoverSprite;
4
5     public Arrow(int sheetIndex) : base("Sprites/LevelObjects/spr_arrow1 @4")
6     {
7         SheetIndex = sheetIndex;
8         normalSprite = sprite;
9         hoverSprite = new SpriteSheet("Sprites/LevelObjects/spr_arrow2 @4", sheetIndex);
10    }
11
12    public override void HandleInput(InputHelper inputHelper)
13    {
14        base.HandleInput(inputHelper);
15        if (BoundingBox.Contains(inputHelper.MousePositionWorld))
16            sprite = hoverSprite;
17        else
18            sprite = normalSprite;
19    }
20 }
```

You could also choose to implement this “hover” functionality in the `Button` class itself. In fact, if you’ve done the exercises from Chap. 18, you may already have an extended `Button` class that supports hovering (and some other features). Feel free to reuse that class here! If an `Arrow` doesn’t have any differences to a `Button` in your case, you could even leave out the `Arrow` class entirely, and just use `Button` instead of `Arrow`.

20.1.2 The *MovableAnimalSelector* Class: Outline

Next, add a class `MovableAnimalSelector` (also in the “LevelObjects” folder). This class should inherit from `GameObjectList`, because it’s going to be an object that contains and manages all four arrows. That way, if we give a `MovableAnimalSelector` a new position, all arrows will automatically move along with it.

The `MovableAnimalSelector` class should keep track of the animal that is currently selected. Also, we’ll use an array to store the four arrow objects, as well as the movement directions that they represent. This results in three member variables:

```
Arrow[] arrows;
Point[] directions;
MovableAnimal selectedAnimal;
```

In the constructor, you can start by defining the four arrow directions:

```
directions = new Point[4];
directions[0] = new Point(1, 0);
directions[1] = new Point(0, -1);
directions[2] = new Point(-1, 0);
directions[3] = new Point(0, 1);
```

These `Point` objects correspond to the following movement directions: right, down, left, and up. Note that this is exactly the order in which the arrow images appear in our sprite sheet.

To create the arrows themselves, add the following code:

```
arrows = new Arrow[4];
for (int i = 0; i < 4; i++)
{
    arrows[i] = new Arrow(i);
    arrows[i].LocalPosition = new Vector2(
        directions[i].X * arrows[i].Width,
        directions[i].Y * arrows[i].Height);
    AddChild(arrows[i]);
}
```

The `LocalPosition` part is interesting: do you see what it does? We’re changing an arrow’s world coordinates based on the values stored in the corresponding `directions` element. That way, the right-pointing arrow will be offset one grid cell to the right, the down-pointing arrow will be offset one grid cell down, and so on.

The final step of the constructor is to initialize the selected animal to `null`. It will help you later on if you add a property for this, which changes the visibility of the selector when you select or deselect an animal. Add the following property:

```
public MovableAnimal SelectedAnimal
{
    get { return selectedAnimal; }
    set
```

```

    {
        selectedAnimal = value;
        Visible = (selectedAnimal != null);
    }
}

```

And in the constructor, set that property to **null**:

```
SelectedAnimal = null;
```

That way, the level will start with an invisible selector. It will become visible as soon as the player selects an animal (but more about that later).

By the way, changing the visibility of *this* object doesn't automatically change the visibility of all *child* objects (the arrows). We'd like to make sure that none of the child objects will be drawn if the selector itself is invisible. In our example code, we've added that functionality to the `GameObjectList` class, because it seems like something that all lists of game objects should do. Hiding a list of objects should mean that you also hide all objects *inside* that list, right? So, go to the `GameObjectList` class, and add the following code at the beginning of its `Draw` method:

```

if (!Visible)
    return;

```

20.1.3 The *MovableAnimalSelector* Class: *Game Loop*

There are two “game loop” methods that `MovableAnimalSelector` should override, namely, `HandleInput` and `Update`. In the `HandleInput` method, you can start by doing nothing if no animal has been selected:

```

if (SelectedAnimal == null)
    return;

```

After that, call the base method:

```
base.HandleInput(inputHelper);
```

This will make sure that all arrows will update their sprites and possibly change their `Pressed` properties. After that, you can check each individual button: if a button has been pressed, the selected animal should move in the corresponding direction. As soon as you've found a pressed button, you can skip the remaining buttons. In short, `HandleInput` should continue as follows:

```

for (int i = 0; i < 4; i++)
{
    if (arrows[i].Pressed)
    {
        SelectedAnimal.TryMoveInDirection(directions[i]);
        return;
    }
}

```

Note: this assumes that the `MovableAnimal` class has a method `TryMoveInDirection`. Add this method now, but leave its body empty. We'll work on that in the next section.

Finally, if the player has clicked anywhere else (so not on any of the four arrows), we'd like the current animal to get deselected. The following code achieves this:

```
if (inputHelper.MouseLeftButtonPressed())
    SelectedAnimal = null;
```

In the Update method, the selector should update its position to match the position of the selected animal. This leads to the following code:

```
public override void Update(GameTime gameTime)
{
    base.Update(gameTime);
    if (SelectedAnimal != null)
    {
        LocalPosition = selectedAnimal.LocalPosition;
        ...
    }
}
```

But let's add one more nice feature: we'll only show the arrows of the direction in which the animal can *actually move*. At the position of "...", add the following code:

```
for (int i = 0; i < 4; i++)
    arrows[i].Visible = SelectedAnimal.CanMoveInDirection(directions[i]);
```

This assumes that `MovableAnimal` has a public method `CanMoveInDirection`. Add that method, and simply let it return **true** for now. In the next section, you'll make the method more advanced, so that it really returns whether or not the animal can move in a particular direction. As a result, the arrows in "blocked" directions will become invisible.

20.1.4 Handling Clicks on an Animal

Next, let's add code that turns an animal into the selected one when the player clicks on it. Start in the `Level` class. When a level gets loaded, this class should add an instance of `MovableAnimalSelector` to the game world. It's useful to store an extra reference to this object as a member variable:

```
MovableAnimalSelector selector;
```

Then, somewhere in the construction process, `Level` should add a selector to the game world and store it in this variable. A nice place to do that is inside the `AddPlayingField` method, just before the last line:

```
selector = new MovableAnimalSelector();
playingField.AddChild(selector);
```

That way, the selector will have the same parent object as the animals. This makes sure that its position will indeed match the selected animal's position.

Also give the `Level` class a `SelectAnimal` method, which takes a `MovableAnimal` instance and sets it as the selected animal of selector:

```
public void SelectAnimal(MovableAnimal animal)
{
    selector.SelectedAnimal = animal;
}
```

Now, it would be nice if any `MovableAnimal` instance can *call* this method when the player clicks on it. This means that `MovableAnimal` should somehow have access to the `Level` object in which it's located. There are many ways to do that, but our proposal is to give the `Animal` class a member variable that refers to the level:

```
protected Level level;
```

The constructor of `Animal` should take a `Level` object as a parameter, and it should store that object in the level member variable:

```
protected Animal(Level level, string spriteName, int sheetIndex = 0)
: base(spriteName, sheetIndex)
{
    this.level = level;
}
```

You'll also have to change the constructors of `MovableAnimal` and `Shark` now, so that they also take a `Level` parameter. Whenever the `Level` class calls these constructors, it should pass itself (**this**) as a parameter. We're confident that you can make these changes yourself now.

With these ingredients in place, a `MovableAnimal` can finally set itself as the selected animal. Go ahead and give that class the following `HandleInput` method:

```
public override void HandleInput(InputHelper inputHelper)
{
    if (Visible && BoundingBox.Contains(inputHelper.MousePositionWorld)
    && inputHelper.MouseLeftButtonPressed())
    {
        level.SelectAnimal(this);
    }
}
```

In other words, an animal asks for itself to be selected when the player clicks on it, but *only* when the animal is currently visible. (Later, we'll make animals invisible when they die or form a pair. So, "visible" can be interpreted as "still actively part of the game.")

20.1.5 The Order of Input Handling

The order in which we draw objects on the screen is important. For example, if we draw the penguins before we draw the background image, the player will never see the penguins.

Until now, we didn't really pay attention to the order in which the `HandleInput` and `Update` methods are called. For instance, in the `GameObjectList` class, this is how we call the `HandleInput` method on all the child objects:

```
foreach (GameObject obj in gameObjects)
    obj.HandleInput(inputHelper);
```

In most cases, this approach works just fine. But in the `Penguin Pairs` game, we may get weird behavior. Suppose that two penguins are next to each other on the playing field and we click on one of the penguins. Then four arrows will appear. Because the two penguins are next to each other, one of the arrows is drawn over the other penguin, as shown in Fig. 20.3. So, if we click on that arrow, what happens? Does the selected penguin move to the left, or do we select the other penguin?

The outcome of this question depends on the order in which input is handled for each game object. If the penguin handles the input before the penguin selector, then the penguin will first mark itself

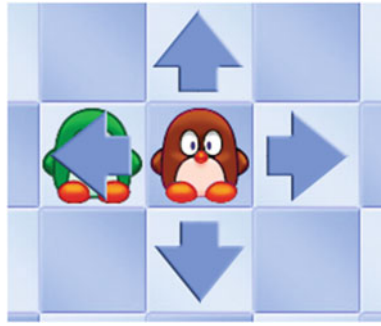


Fig. 20.3 What happens when we click on the left arrow?

as the new selected animal. A bit later, it's the turn of the selector, and it will realize that the left arrow has been pressed. As a result, the *newly selected* penguin will start moving! That is not what's supposed to happen. Instead, the desired behavior is that the *initially selected* penguin moves.

How can you make sure that the desired behavior occurs? Well, the easiest solution (for now) is to make `GameObjectList` let its child objects handle input in the opposite order. That way, objects that are drawn on top will handle input first. You can easily do this with the following **for** loop:

```
for (int i = children.Count - 1; i >= 0; i--)
    children[i].HandleInput(inputHelper);
```



CHECKPOINT: PenguinPairs4a

Compile and run the game. You should now be able to click on a penguin or seal to select it. Four arrows will then be shown around the selected animal.

20.2 Making the Animals Move

In this section, you'll add the code that lets animals move around when the player asks for it. You won't be dealing with the real *logic* of the game yet, such as forming pairs between penguins; that will be the topic of the *next* section. For now, we'll focus on the basic ingredients for moving animals in certain directions.

20.2.1 Linking the Animals to the Grid

Before looking into the `MovableAnimal` class, it's useful to change the program so that each animal keeps track of its own current position in the grid. If this position changes, then the animal will tell that to the grid and change the necessary data. To prepare for this, give the `Level` class two convenient methods for adding or removing an animal at a grid position:

```
public void AddAnimalToGrid(Animal animal, Point gridPosition)
{
    animalsOnTiles[gridPosition.X, gridPosition.Y] = animal;
}
```

```
public void RemoveAnimalFromGrid(Point gridPosition)
{
    animalsOnTiles[gridPosition.X, gridPosition.Y] = null;
}
```

Next, give the `Animal` class another member variable that stores an animal's grid position:

```
protected Point currentGridPosition;
```

It makes sense to give the constructor a new matching parameter, too:

```
protected Animal(Level level, Point gridPosition, string spriteName, int sheetIndex = 0)
    : base(spriteName, sheetIndex)
{
    this.level = level;
    currentGridPosition = gridPosition;
}
```

Again, you'll also have to change the constructors of `MovableAnimal` and `Shark` now, just like when you added the `Level` parameter earlier.

To put an animal in charge of adding itself to the grid, give the `Animal` class a method `ApplyCurrentPosition`. This method should call the `AddAnimalToGrid` method that you've just created. Also, we'll let this method give the animal the correct physical position in the game world. In total, the method looks like this:

```
protected virtual void ApplyCurrentPosition()
{
    level.AddAnimalToGrid(this, currentGridPosition);
    LocalPosition = level.GetCellPosition(currentGridPosition.X, currentGridPosition.Y);
}
```

We've marked this method as **virtual**, because later, `MovableAnimal` will receive an advanced version that does more work.

Call this new method at the end of your `Animal` constructor. That way, when you create a new instance of an animal, this animal will automatically give itself the correct world position *and* add itself to the grid. This means that `Animal` has now taken over some of the work that `Level` used to do. In the `AddAnimal` method of the `Level` class, you should now remove the last `if` block (starting with `if (result != null)`), because the `Animal` constructor already does these tasks now.

20.2.2 Overview of the *MovableAnimal* Design

Okay, so all animals store their position in the grid, and they can notify the grid when this position changes. We should now look at `MovableAnimal`: the class that represents an animal that can actually move. This class already has an empty method `TryMoveInDirection`, but it's not yet capable of really moving to another cell.

Let's fill in this class further now, using the following overall design:

1. Just like back in the `Jewel` class, each `MovableAnimal` instance will have a *target position* in the game world, and it will move to that position at a certain speed. To give the illusion that the animals are sliding on ice, we'll let the animals move at a constant speed, until they've reached the target position.

2. When the player tells the animal to move in a direction, the animal will keep adjusting its `currentGridPosition` by one step in that direction until it's no longer possible. When this process is over, the final value of `currentGridPosition` stores the grid position that the animal should move to. This also leads to a new target position in the game world.
3. When the animal has just calculated a new target position and starts moving, it *removes* itself from the level grid.
4. When the animal has reached its destination, it will *add* itself to the grid again, using the `ApplyCurrentPosition` method. We'll give `MovableAnimal` a special version of this method that does some extra checks. Depending on the type of tile where the animal has ended up, this method might mark the animal as "stuck inside a hole," or it might make the animal disappear.

You'll now implement these items one by one, starting with the concept of a "target position." Give the `MovableAnimal` class two more member variables:

```
Vector2 targetWorldPosition;
const float speed = 300;
```

In the constructor, set `targetWorldPosition` to the value of `LocalPosition`. That way, the animals will stand still by default. It's also useful to add a property that checks if the animal currently has a different position to go to:

```
bool IsMoving { get { return LocalPosition != targetWorldPosition; } }
```

20.2.3 Moving and Stopping

The `TryMoveInDirection` method should be in charge of sending an animal in a certain direction. This corresponds to items 2 and 3 of our list. Let's fill in this method now.

First, if the animal cannot move in the requested direction at all, then we can immediately jump out of the method again:

```
if (!CanMoveInDirection(direction))
    return;
```

Otherwise, the animal is about to move. Start by *removing* the animal from its current position in the grid, as we've explained in item 3:

```
level.RemoveAnimalFromGrid(currentGridPosition);
```

Next, as explained in item 2, we should push `currentGridPosition` into the given direction as much as possible. That is, the animal should move in that direction as long as it can do so:

```
while (CanMoveInDirection(direction))
    currentGridPosition += direction;
```

After the loop has finished, `currentGridPosition` stores the animal's new grid cell. Use it to calculate a new target position in the game world:

```
targetWorldPosition = level.GetCellPosition(currentGridPosition.X, currentGridPosition.Y);
```

Finally, calculate a velocity that sends the animal towards that point:

```
Vector2 dir = targetWorldPosition - LocalPosition;
dir.Normalize();
velocity = dir * speed;
```


This concludes the `TryMoveInDirection` method. Next, let's think about what it means if an animal *stops* moving. When an animal has reached its target position, it should call `ApplyCurrentPosition` to tell the grid that the animal has now reached a particular grid cell. To implement this, override the `Update` method:

```
public override void Update(GameTime gameTime)
{
    base.Update(gameTime);
    if (IsMoving && Vector2.Distance(LocalPosition, targetWorldPosition)
        < speed * gameTime.ElapsedGameTime.TotalSeconds)
    {
        ApplyCurrentPosition();
    }
}
```

The somewhat complicated `if` condition checks if the animal will reach its goal before the next frame. Stated differently, it checks if the animal would “overshoot” its goal if it continued moving for one more frame. If that happens, we'll act as if the animal has reached its goal.

You've now written the code that lets an animal start and stop moving when the player clicks on an arrow. You *could* compile and run the game now, but you'll see that the game will freeze as soon as you click on a blue arrow, which isn't really what we want.¹ This happens because the `CanMoveInDirection` method always returns `true`, which results in an infinite loop in our `TryMoveInDirection` method.

20.3 Adding the Game Logic

At this point, the basic ingredients of movement are here, but we still need to give the animals the correct behavior so that the levels become playable. In this section, you'll implement this *game logic* by filling in the remaining gaps. In particular, you'll fill in the `CanMoveInDirection` method, and you'll create `MovableAnimal`'s custom version of the `ApplyCurrentPosition` method.

At the end of this section, you'll finally be able to move penguins and seals around in the correct way! However, just like with the game logic of *Jewel Jam*, this step involves quite a bit of code. Use the *PenguinPairs4b* example project if you get confused somewhere.

20.3.1 Preparing the *Level* Class

To implement our game logic, it's convenient to give the `Level` class some methods that return the *tile type* at a certain grid position and the *animal* at a certain grid position (which might be null). Both methods will need to check if a given position lies inside the grid, so let's add a helper method for that first:

```
bool IsPositionInGrid(Point gridPosition)
{
    return gridPosition.X >= 0 && gridPosition.X < GridWidth
    && gridPosition.Y >= 0 && gridPosition.Y < GridHeight;
}
```

¹ Although freezing *does* match the game's icy theme.

Here, `GridWidth` and `GridHeight` are read-only properties that are nice “shortcuts” for getting the width and height of the tiles array. Add these properties as follows:

```
int GridWidth { get { return tiles.GetLength(0); } }
int GridHeight { get { return tiles.GetLength(1); } }
```

With these ingredients, we can create the methods for returning the tile type and animal in a cell. The `GetTileType` method below returns the tile type at a given grid position. If the given position lies outside the grid, then this method returns the `Empty` type. This makes sense: the level is surrounded by water, so why not treat non-existing tiles that way?

```
public Tile.Type GetTileType(Point gridPosition)
{
    if (!IsPositionInGrid(gridPosition))
        return Tile.Type.Empty;
    return tiles[gridPosition.X, gridPosition.Y].TileType;
}
```

Add this method to the `Level` class. Similarly, add a `GetAnimal` method that returns the animal at a given grid position (which might be `null`, but that’s okay). If the position lies outside the grid, the method should return `null`.

20.3.2 Checking if a Move Is Possible

With these helper methods, we can finally give `MovableAnimal` the correct behavior. First, let’s fill in the `CanMoveInDirection` method. This method should return whether or not an animal can move *one step* in a certain direction. Remember: the `TryMoveInDirection` method will keep calling this method (in a **while** loop) until movement is no longer possible. So, indirectly, it’s up to `CanMoveInDirection` to decide when an animal needs to stop moving.

Currently, this method currently returns **true** all the time. Instead, it should return **false** if the animal cannot move any further (for any reason) and **true** otherwise.

To start off, there are three cases in which an animal definitely cannot move, regardless of the direction we’re asking for. This happens if:

- the animal is currently already moving, stuck inside a hole, or invisible;
- the grid cell at `currentGridPosition` stores an empty tile or a hole tile;
- the grid cell at `currentGridPosition` already stores *another animal*. This other animal could be a shark, or an animal that forms a pair, but that doesn’t matter here yet. All we need to know is that movement should stop.

In any of those cases, the method should return **false**. Implement these cases yourself, using the `GetTileType` and `GetAnimal` method that you’ve added to the `Level` class earlier. (Watch out: “another animal” means that `GetAnimal` returns an `Animal` object that isn’t `null`, and that also isn’t equal to **this**.)

If none of these cases occur, then the animal is at least allowed to leave its current cell. It’s then time to look at the *next* grid cell, taking the direction parameter into account:

```
Point nextPosition = currentGridPosition + direction;
```

If the grid cell at `nextPosition` stores a *wall tile*, then the animal shouldn’t be allowed to reach that tile, and the movement should stop now. Go ahead and implement this case.

If the grid cell at `nextPosition` stores an *animal*, then the behavior should depend on what *kind* of animal that is. This step is a bit too difficult to implement right now, but we can already think about the behavior that we'd like to see:

- If the animal at `nextPosition` is a seal, or if it's a penguin that doesn't have a matching color, then that animal should basically act as a wall. Our method should return **false** here, to indicate that the moving animal must stop.
- If the animal at `nextPosition` is a shark or a penguin with a matching color, then movement *is* allowed. This move will cause our animal to disappear (because it will either die or form a pair), but that doesn't matter as far as the `CanMoveInDirection` method is concerned. All we care about right now is that the move is allowed.

So, only the *first* option is a special case in which we want to return **false**. But how should we implement that case? How can we determine if two `MovableAnimal` objects can form a pair? And how can we even check if an `Animal` is a `MovableAnimal` (and not a `Shark`)? We'll answer those questions in the next subsections.

Assuming that we've filled in this missing case, there are no other events that could block the animal's movement. So, at the very end of this method, you can return **true**, indicating that the animal can move in the given direction. It doesn't matter what the result of this move will be: it could lead the animal into the water, into the mouth of a shark, onto a matching penguin, or just to an ordinary tile that allows yet another step of movement. But for the *current* call to `CanMoveInDirection`, all that matters is that at least *one* step of movement is possible. Whatever happens after that will be discovered in the *next* call to the same `CanMoveInDirection` method.

20.3.3 Checking if Two Movable Animals Can Form a Pair

Now, let's work on the final difficult case that we promised to get back to. To check if one `MovableAnimal` instance can *form a pair* with another `MovableAnimal` instance, start by giving the `MovableAnimal` class two convenient properties:

```
bool IsSeal { get { return AnimalIndex == 7; } }
bool IsMultiColoredPenguin { get { return AnimalIndex == 6; } }
```

Next, add a helper method with the following header:

```
bool IsPairWith(MovableAnimal other)
```

This method should return whether or not the current animal (**this**) can form a pair with the animal indicated by `other`. If you think about it, there are three cases:

- If either of the two animals is a seal, then the method should return **false**. After all, a seal can never form a pair with another animal: a seal essentially just a wall that can be moved around.
- Otherwise, both animals are definitely penguins. If either of them is a multicolored penguin, then the method should return **true**. After all, a multicolored penguin can form a pair with any other penguin.
- Otherwise, the two penguins can form a pair if they have the same color (i.e., the same `AnimalIndex`).

Use this description to write the `IsPairWith` method yourself.

20.3.4 Checking if an Animal Is Movable: The Keyword *is*

To implement the final `CanMoveInDirection` case, first retrieve the animal at the next cell:

```
Animal nextAnimal = level.GetAnimal(nextPosition);
```

After this instruction, the `nextAnimal` variable will store either **null** or an `Animal` instance. But this instance will always be something more specific than just an `Animal`: it will be a `MovableAnimal` or a `Shark`.

To be able to use our new `IsPairWith` method, we have to give that method a `MovableAnimal` and not just an ordinary `Animal`. As a first attempt, you could try to *cast* the `nextAnimal` object to a `MovableAnimal`. The code would then look like this:

```
if (nextAnimal != null && !IsPairWith((MovableAnimal)nextAnimal))  
    return false;
```

This seems correct: we first check if `nextAnimal` really exists, and then we use the `IsPairWith` method to compare two `MovableAnimal` objects.

But what if `nextAnimal` is actually a `Shark`? Then we have a problem: casting a `Shark` to a `MovableAnimal` is impossible, because they aren't subclasses of each other! As a result, your program will crash, just like (for example) when you try to cast a **string** to an **int**.

Instead of doing a cast and hoping for the best, it would be nice if we could check if the `Animal` object returned by `GetAnimal` is actually an instance of `MovableAnimal`. It turns out that C# offers a nice keyword for this exact purpose. The following code fragment:

```
nextAnimal is MovableAnimal
```

is a Boolean expression that has the value **true** if `nextAnimal` is indeed a `MovableAnimal` instance and **false** if it is not. In other words, the keyword “**is**” checks if a cast to `MovableAnimal` would be safe or not. If the result is **true**, then you can be sure that this cast won't crash the program.



Quick Reference: The Keyword *is*

Let `MyClass` be any class name in your program, and let `myObject` be any object or primitive. The following expression:

```
myObject is MyClass
```

returns whether or not `myObject` can safely be cast to the `MyClass` type. This is **true** when `myObject` is an instance of `MyClass` or of any *subclass* of `MyClass`. It is **false** in all other cases: that is, if `myObject` is **null** or if it has a completely unrelated type.

As you can see in this Quick Reference box, the keyword “**is**” can also deal with a value of **null**. So, if `nextAnimal` has the value **null**, then the expression `nextAnimal is MovableAnimal` returns **false** as well. In other words, we can rewrite our previous attempt as follows:

```
if (nextAnimal is MovableAnimal && !IsPairWith((MovableAnimal)nextAnimal))  
    return false;
```

With this version, your program will not crash. Why does this work? Well, the components of a Boolean expression are actually checked from left to right. As soon as the program knows the value of the entire expression, it will stop checking the remaining components. Here, our **if** instruction will *first* check the “nextAnimal **is** MovableAnimal” part. If that part already isn’t true, then the program won’t even bother to check the “IsPairWith(...)” expression, because it already knows that the **if** check will fail anyway. As a result, the cast to MovableAnimal will only be performed if we know that it’s safe.

Lazy Evaluation — This idea of treating parts of an expression one by one is called **lazy evaluation**. Many programming languages use this concept, because it can often make a program more efficient.

In this example, if you write the “IsPairWith(...)” part first and the “**is** MovableAnimal” part second, the program will still crash when the cast fails! This can be confusing: logically speaking, it shouldn’t matter if you flip the two parts around a **&&** symbol. “A and B” should mean exactly the same as “B and A,” right?

Well, yes, they *mean* the same thing, but in a C# program, a different order can still have different effects, due to the order in which things are calculated. Forgetting about this difference usually isn’t a problem, but it can become tricky if the success of one part relies on the success of another part.

With the CanMoveInDirection method now finally finished, the TryMoveInDirection method doesn’t freeze the game anymore. Feel free to compile and run the game again to try it out. You’ll be able to move penguins and seals around, and you’ll see that the blue arrows are only shown for movement directions that are actually allowed. That’s because the MovableAnimalSelector class uses the same CanMoveInDirection method that you’ve just filled in. However, it’s not yet possible to really create a pair of penguins, and penguins cannot die or get stuck in holes yet. This is because we’re still missing one more method.

20.3.5 Applying the New Position of a Movable Animal

One last method remains to be written: the custom version of ApplyCurrentPosition for movable animals. As you know, this method gets called when a MovableAnimal has reached its target position. Currently, only the standard Animal class implements this method: it sets the animal’s position in the game world and then stores the animal in the grid at currentGridPosition. However, MovableAnimal’s version of ApplyCurrentPosition should do a lot more. Here’s a list of the tasks that this method should do (in this order):

- Set the animal’s LocalPosition to the world position that matches the current cell, just like in Animal’s version of the method.
- Set the velocity to zero, so that the animal stops moving.
- Instead of adding the animal to the grid at its new position, first check if the grid cell at currentGridPosition has the Empty tile type. If so, then this animal should now fall into the water. It should become invisible and not add itself to the grid anymore.
- Otherwise, check if the current grid cell is already occupied with another animal. If so, then that animal is a shark or a matching penguin. (It can never be a seal or a nonmatching penguin, because then our animal should have stopped moving earlier!) In both cases, both involved animals should become invisible, and the grid cell should become empty.

- Otherwise, the animal stays in the game. It should add itself to the grid at its new position, using the `AddAnimalToGrid` method.
- If the animal stays in the game and its grid cell has the `Hole` type, then the animal should set its `isInHole` property to **true**. This will automatically change the animal's sprite.

Based on these hints, try to write the `ApplyCurrentPosition` method yourself. It's easy to make a mistake, though, and even small errors could lead to confusing bugs in the game. If you get stuck, you can also copy the method from our `PenguinPairs4b` project.

20.3.6 Cleaning Up the Code

Something interesting has happened now: the `ApplyCurrentPosition` method automatically sets the `isInHole` property to **true** if the animal is in a hole tile. But `ApplyCurrentPosition` is also called in the *constructor* of each animal, when the level starts. The constructor of `MovableAnimal` currently has a `isInHole` parameter, but this has now become redundant: indirectly, the constructor will already check by itself whether or not the animal is in a hole. So, you can now simplify the `MovableAnimal` constructor as follows:

```
public MovableAnimal(Level level, Point gridPosition, int animalIndex)
: base(level, gridPosition, GetSpriteName(false), animalIndex)
{
    targetWorldPosition = LocalPosition;
}
```

We're now using **false** as a standard value for `GetSpriteName`, to indicate that an animal is *not* inside a hole by default. This will initialize each penguin and seal with a "non-holed" sprite. If the animal starts at a hole tile, then the `ApplyCurrentPosition` method will recognize that, and it will make the correct changes to this animal. The animal will then immediately "fall" into the hole at its starting position.

Now that the `MovableAnimal` constructor no longer has a `isInHole` parameter, you'll notice that the `AddAnimal` method of the `Level` class can be simplified even further. Try to simplify this method yourself, or take a look at our solution in the `PenguinPairs4b` project.



CHECKPOINT: PenguinPairs4b

Compile and run the game. You can now move penguins and seals, and they will behave correctly.

The game is now very close to playable! Penguins and seals can be moved around, they can get stuck inside holes, and they will disappear from the board when they meet a shark or a matching animal. However, some parts are still clearly missing. The game doesn't keep track of how many pairs the player has made, so it cannot detect when the player has finished a level. We'll fix that in the next section. Also, the "hint" functionality doesn't work yet, and there's no way to restart a level without going back to the menu. We'll leave those finishing touches for the next chapter.

20.4 Maintaining the Number of Pairs

The goal of each level is to create a certain number of penguin pairs. This section lets you add a game object that shows how many pairs the player has made so far. When the level starts, this object will show a list of gray penguin shapes, one for each pair that the player should make. Each time the player creates a pair, a gray penguin will be colored in.



Fig. 20.4 The sprite containing all the possible pair images

For the gray and colored penguins of this object, we’ve prepared another sprite sheet (*spr_penguin_pairs*), which is also shown in Fig. 20.4. Note that the colors appear in the exact same order as in the other sprite with penguins and seals. This will turn out to be very important! The final sprite in this sheet plays the role of an “empty” pair that the player hasn’t found yet. This is OK because it’s not possible to create pairs of *seals*, so the eighth sprite in the sheet is now available for something else.

Add this sprite to the project, as well as *spr_frame_goal* (which is the background image that we’ll draw behind all penguin shapes).

20.4.1 The *PairList* Class

Let’s create a class *PairList* for this new game object. Because this class is rather simple, we’ll just give you the source code instead of talking you through the creation process. Listing 20.2 shows the full *PairList* class; you can also find it in the *PenguinPairs4c* project. Go ahead and add it to the “LevelObjects” folder of your project.

Listing 20.2 The *PairList* class

```

1 using Microsoft.Xna.Framework;
2
3 class PairList : GameObjectList
4 {
5     int nrPairsMade;
6     SpriteGameObject[] pairObjects;
7
8     public PairList(int nrPairs)
9     {
10         // add the background image
11         AddChild(new SpriteGameObject("Sprites/spr_frame_goal"));
12
13         // add a sprite object for each pair that the player should make
14         Vector2 offset = new Vector2(100, 7);
15         pairObjects = new SpriteGameObject[nrPairs];
16         for (int i = 0; i < nrPairs; i++)
17         {
18             pairObjects[i] = new SpriteGameObject("Sprites/spr_penguin_pairs@" + i, 7);
19             pairObjects[i].LocalPosition = offset + new Vector2(i * pairObjects[i].Width, 0);
20             AddChild(pairObjects[i]);
21         }
22
23         // start at 0 pairs
24         nrPairsMade = 0;
25     }
26
27     public void AddPair(int penguinIndex)
28     {
29         pairObjects[nrPairsMade].SheetIndex = penguinIndex;
30         nrPairsMade++;
31     }
32
33     public bool Completed { get { return nrPairsMade == pairObjects.Length; } }
34 }

```

Here's an overview of what the class does:

- The class inherits from `GameObjectList`, because it's essentially a list of smaller objects (plus some extra data).
- The class has two extra member variables: the number of pairs that the player has made so far and an array with convenient references to each penguin sprite.
- The constructor method first adds a background sprite, and then it adds a penguin image for each pair that the player should make. Each penguin starts out with a sheet index of 7, which corresponds to the gray penguin sprite. The *total number of pairs* is a parameter of the constructor: this is useful because that number can be different for each level.
- The `AddPair` method turns the first gray penguin shape into another color, by changing that object's sprite sheet index. It also increments the `nrPairsMade` counter, so that the *next* call to `AddPair` will change the *second* penguin shape and so on.
- The `Completed` property checks if `nrPairsMade` matches the total number of penguin images in the list. If that's true, then the player has made all required pairs, and he/she has finished the level. In the next chapter, you'll use this property to let players go to the next level.

20.4.2 Storing and Showing Pairs in the Level

The `Level` class should create an instance of this new `PairList` class when the level gets loaded. A good place to do this is inside the `AddLevelInfoObjects` method that you already have. This method currently adds the title and description to the bottom of the screen. To let it create a `PairList` as well, add the following instructions:

```
pairList = new PairList(targetNumberOfPairs);
pairList.LocalPosition = new Vector2(20, 20);
AddChild(pairList);
```

As usual, make sure that `pairList` is a member variable that stores an extra reference to this object. This is useful because you'll want to refer to this object later on. After all, when the player creates a new pair, you'll want to call the `AddPair` method for this specific object.

Remember: `targetNumberOfPairs` is a member variable of `Level`, so this code will only work if you've already given that variable a value *before* entering the `AddLevelInfoObjects` method. This may not yet be the case in your current code. In the `PenguinPairs4c` project, we've moved the method call to `AddLevelInfoObjects` down a bit; make sure to do the same yourself, if necessary.

Next up, how can we notify the level that a new pair has been found? First, give the `Level` class the following public method:

```
public void PairFound(MovableAnimal penguin1, MovableAnimal penguin2)
{
    int penguinType = MathHelper.Max(penguin1.AnimalIndex, penguin2.AnimalIndex);
    pairList.AddPair(penguinType);
}
```

The first instruction in this method calculates the sprite sheet index (the color) of the pair that needs to be added to the list. Usually, this will be the same color as the penguins in the method's parameters. But if one of the two penguins is a multicolored (rainbow) penguin, we'd always like to show the rainbow image. In Fig. 20.4, the rainbow penguin comes after all regular penguins. This means that it's safe to always use the *largest* of the two sprite sheet indices from both penguins in the pair. The `MathHelper.Max` function (which is part of the MonoGame engine) does exactly that.

The second instruction calls the `AddPair` method with the sprite sheet index that you’ve just calculated. As a result, a gray penguin from the `PairList` will change its color.

20.4.3 Adding a Pair at the Right Time

The final step is to let a game object *call* this `PairFound` method at the right time. The best place to do this is in the `MovableAnimal` class, inside the `ApplyCurrentPosition` method.

This method already makes the current `MovableAnimal` instance invisible when it reaches a tile with another animal. You’ve written this part of code yourself, so we can’t say for sure what it looks like in your case. In our example projects, the code looks like this:

```
Animal otherAnimal = level.GetAnimal(currentGridPosition);
if (otherAnimal != null)
{
    level.RemoveAnimalFromGrid(currentGridPosition);
    Visible = false;
    otherAnimal.Visible = false;
    return;
}
```

If the “other animal” is *not* a shark, then we’ve found a pair! So, just before the **return** statement here, add the following code to call the `PairFound` method:

```
if (otherAnimal is MovableAnimal)
    level.PairFound(this, (MovableAnimal)otherAnimal);
```

This is another example of using the keyword “**is**”, to check if the other animal is actually a `MovableAnimal` (and not a `Shark`). As we’ve explained before, the other animal *must* be either a shark or a matching penguin. Otherwise, the current animal shouldn’t even be allowed to reach this tile in the first place.



CHECKPOINT: PenguinPairs4c

Compile and run the game again. If you create a pair of penguins now, then one of the gray penguin shapes in the corner should receive a color.

This concludes the chapter. Gameplay-wise, the Penguin Pairs game is now as good as finished. In the next chapter, you’ll add finishing touches to the game, including the possibilities to complete or reset a level.

20.5 What You Have Learned

In this chapter, you have learned:

- how to handle the selection of game objects;
- how to model interactions between different kinds of game objects;
- how to use the keyword **is** to check if an object has a certain type.

20.6 Exercises

1. The Keyword “**is**”

Assume that we have a game with an abstract class `Animal`, and several classes that inherit from it: `Dog`, `Cat`, and `Fox`. Now consider the following method, which takes a list of `Animal` objects and lets all animals in that list say something.

```
public void LetAnimalsSpeak(List<Animal> animals)
{
    foreach (Animal animal in animals)
    {
        // ...
    }
}
```

In this question, you’ll fill in this method in two different ways.

- Inside the **foreach** loop, add code that checks if the current animal in the loop is a `Dog`, a `Cat`, or a `Fox`. Use the keyword “**is**” that we introduced in this chapter. For each type of animal, write a different line of text to the console, using the `Console.WriteLine` method. For example, you could print the text “Woof!” if the animal is a `Dog` and “Meow!” if the animal is a `Cat`.²
- Another solution is to give the `Animal` class an abstract method `Speak` that each specific animal needs to override. Implement this solution. *Hint:* We’ve done something like this in an earlier chapter already.
- From a software design point of view, why is the second solution better than the first?

²You can decide for yourself what the `Fox` says.

Chapter 21

Finishing the Game



In this chapter, you'll finalize the Penguin Pairs game. You'll add the final missing features, sound effects, and background music. Finally, we'll show you how to move the general "game engine" code to a separate project (a *library*), to make it even easier to reuse.

21.1 Hints and Resetting

This section lets you add two remaining features of the game: showing hints and resetting a level. These features are not very difficult to add. We'll give a very short overview of what you need to do, and we'll let you fill in the details yourself. If you ever get stuck, have a look at the PenguinPairs5a example project.

21.1.1 Showing a Hint

The first feature you're going to add is the hint functionality. When the player is playing a level and clicks on the "hint" button in the top-right corner, we'd like to make the hint arrow visible for 2 s.

The `Level` class already has a member variable `hintArrow` that refers this arrow, and it already makes sure that the arrow is initially invisible. To make this arrow (briefly) visible when the "hint" button gets clicked, do the following:

- Give the `Level` class another member variable: `VisibilityTimer hintTimer`. Just after you add the hint arrow to the game world, also add a `VisibilityTimer` to the game world (with `hintArrow` as its target) and store a reference to it in the `hintTimer` variable.
- Give the `Level` class a method **public void** `ShowHint()` that lets `hintTimer` make the arrow visible for 2 s.
- In the `PlayingState` class, call this method when the player has clicked on the "hint" button.

But the hint only says something about the *first* move that the player should make. So, the hint arrow will probably not make sense anymore if the player has already made a move. Therefore, make the following changes as well:

- Give the `Level` class a property **bool** `FirstMoveMade`, with a public **get** component and a private **set** component. Give it an initial value of **false**.

- Set this property to **true** when the player has made their first move. A good place for this is the `RemoveAnimalFromGrid` method: this method gets called when an animal starts moving, and this can only happen in response to a move by the player.
- The “retry” button should be visible whenever the `FirstMoveMade` property is **true**. The “hint” button should be visible whenever the `FirstMoveMade` property is **false** and the hint functionality is enabled. This leads to the following code that you can add to the `Update` method of `PlayingState`:

```
hintButton.Visible = PenguinPairs.HintsEnabled && !level.FirstMoveMade;
retryButton.Visible = level.FirstMoveMade;
```

You can now also remove the line from `LoadLevel` that makes the “hint” button visible or invisible when the level is loaded. After all, we’re now setting the visibility of the “hint” button in each frame of the game loop.

Compile and run the game again, and verify that the “hint” button now gets replaced by the “retry” button after the first move. We’ll work on that button next.

21.1.2 Resetting a Level

When the player clicks on the “retry” button, we’d like to reset the level to its initial state. This means that all animals need to go back to their starting position and that they should reappear if they were invisible. Furthermore, the number of found pairs should be reset to zero, and the “hint” button should reappear instead of the “retry” button.

To implement this, it’s a good idea to use the `Reset` method that all game objects already have. We just need to think about *what* each object should actually do when it resets itself. These details are a bit tricky to get right, and (as usual) there’s not one perfect solution. We’ll show you one possibility, but you’re free to try out other solutions as well.

In the `PlayingState` class, start by calling `level.Reset()` when the “retry” button has been clicked. By now, it should be obvious where you can write the code for this.

Next, give the `Level` class its own `Reset` method. The method should look like this:

```
public override void Reset()
{
    for (int y = 0; y < GridHeight; y++)
        for (int x = 0; x < GridWidth; x++)
            animalsOnTiles[x, y] = null;

    FirstMoveMade = false;
    base.Reset();
}
```

This method first clears the `animalsOnTiles` array. This makes the level think that there are no more animals on the grid. Next, it sets the `FirstMoveMade` property back to **false**, to make sure that the “hint” button will be shown instead of the “retry” button. Finally, it calls `base.Reset()`. This will execute `GameObjectList`’s version of the `Reset` method, which will (in turn) call the `Reset` method of all game objects in the list.

Note: because of the last instruction, this method relies on *other* objects to reset themselves in the correct way. So, our next task is to give other classes in the game their own `Reset` method and to think about what resetting really means in their case.

21.1.3 Resetting the Other Game Objects

It turns out that most classes don't need any special resetting behavior, so they can just use the `Reset` method that they already inherit. In fact, there are only four classes that need their own custom `Reset` method: `MovableAnimalSelector`, `PairList`, `Animal`, and `MovableAnimal`. The first three of these classes should reset themselves as follows:

- `PairList` should reset the number of found pairs to zero, and it should reset the sprite sheet index of all penguin images to 7 (the gray penguin shape).
- `MovableAnimalSelector` should set the `SelectedAnimal` property to `null`, so that no animal is selected when the level restarts.
- `Animal` should make itself visible again, and it should call `ApplyCurrentPosition` to reinsert itself into the grid.

See if you can write these `Reset` methods yourself. Also make sure to call `base.Reset()` in all cases. Strictly speaking, this is not always necessary, but it doesn't hurt either. If you want to see our code, have a look at the `PenguinPairs5a` project.

The trickiest class to reset is `MovableAnimal`. When a `MovableAnimal` resets itself, it should go back to its starting position in the grid. To make this possible, give the `MovableAnimal` class a new member variable, which will store a backup of the animal's starting position:

Point `startPosition`;

Initialize this variable in the constructor:

```
startPosition = gridPosition;
```

The `Reset` method of `MovableAnimal` should then look like this:

```
public override void Reset()
{
    currentGridPosition = startPosition;
    IsInHole = false;

    base.Reset();

    targetWorldPosition = LocalPosition;
}
```

This method first reverts the animal to its starting position, and it marks the animal as “no longer stuck inside a hole.” *After that*, it calls `base.Reset()`, which will execute `Animal`'s version of the `Reset` method. That method will re-add the animal to the grid, and it will give the animal the correct position in the game world. The final step for `MovableAnimal` is to reset its *target* position; that way, the animal will no longer try to move.

If you put these instructions in a different order, strange things may happen. It might be a nice exercise to change the order on purpose and to try and understand what happens in the game then.

Compile and run the game again. The “retry” button should now behave correctly.

21.2 Completing a Level

In this section, you'll add all functionality related to *completing* a level. As soon as the player has created enough penguin pairs, we want to show an image saying that the level has been completed. We also want to *store* the fact that the level is now completed, so that the player doesn't have to complete

it again next time. For this, we can use the `MarkLevelAsSolved` method that we've already prepared a few chapters ago. Finally, we want to send the player to the next level when he/she presses the left mouse button.

Let's put the `PlayingState` class in charge of handling all these things.

21.2.1 Updating the *PlayingState* and *Level* Classes

Start by including the sprite *spr_level_finished* into the project. This is a full-screen image containing the overlay that players should see when they finish a level. In the constructor of `PlayingState`, add this overlay as a `SpriteGameObject` after the buttons. Store an extra reference to it in a member variable named `completedOverlay`.

In the `LoadLevel` method of `PlayingState`, make the overlay invisible. This will hide the overlay each time the player starts playing a new level.

To show the overlay at the right time, let's give `PlayingState` a new method that we'll call when the player completes a level. This method should *show* the overlay, *hide* the level, and call the `MarkLevelAsSolved` method to update the player's progress:

```
public void LevelCompleted(int levelIndex)
{
    completedOverlay.Visible = true;
    level.Visible = false;

    PenguinPairs.MarkLevelAsSolved(levelIndex);
}
```

Note: we have to hide the level because the overlay was added *before* the level object itself. If we did not hide the level, the overlay would accidentally be drawn behind the level. It's a bit unfortunate that we have to make the level invisible, but it's the easiest solution for now.

In the next part of this book, we'll extend the game engine so that sprites and text can be drawn at a certain *depth*. That way, it no longer matters in what order the elements were added to the game world.

Anyway, when should this `LevelCompleted` method get called? It makes sense to do this when the player creates the final required penguin pair in a level. So, go to the `PairFound` method of the `Level` class. After you've updated the `pairList` object, add code that calls the `LevelCompleted` if the level is now indeed completed:

```
if (pairList.Completed)
{
    PlayingState playingState = (PlayingState)ExtendedGame
        .GameStateManager.GetGameState(PenguinPairs.StateName_Playing);
    playingState.LevelCompleted(LevelIndex);
}
```

21.2.2 Going to the Next Level

Next up, let's change the game so that the player can immediately go to the next level when a level has been solved. Start by giving the main `PenguinPairs` class the following method:

```

public static void GoToNextLevel(int levelIndex)
{
    if (levelIndex == NumberOfLevels)
        GameStateManager.SwitchTo(StateName_LevelSelect);

    else
    {
        PlayingState playingState =
            (PlayingState)GameStateManager.GetGameState(StateName_Playing);
        playingState.LoadLevel(levelIndex + 1);
    }
}

```

This method sends the player to the next level, using the index of the *current* level as a parameter. If the current level is the last level, then there isn't a next level to go to. In that case, the player should be sent back to the level selection screen. In all other cases, the *PlayingState* class should load the next level. We don't have to switch to a different game state then, because *PlayingState* is already the active state.

We want to call this method when the player has finished a level and then presses the left mouse button. To implement that, go to the *HandleInput* method of *PlayingState*. If the *completedOverlay* image is visible (meaning that the player has finished the level), check if the player has pressed the left mouse button. If that's true, call the *GoToNextLevel* method.

The remaining code of *HandleInput* (calling *level.HandleInput* and checking if the UI buttons have been pressed) should only happen when the overlay image is *not* visible. You can implement that by combining **if** and **else** instructions in the right way. Take a look at the *PenguinPairs5a* example project if you want to see a possible solution.

21.2.3 Updating the Buttons in the Level Menu

A final detail is that the buttons on the level selection screen should change when the status of a level changes. The *LevelButton* class currently has a *LevelStatus* member variable. At all times, the value stored in this variable *should* be equal to the status of the corresponding level. However, when the player has just finished a level (and unlocked the next level), there is a mismatch between the status of the *level* and the status of the *button*. As a result, the level selection screen might look wrong when the player goes back to it.

There are many ways to solve this mismatch. One possible solution is to give the *LevelMenuState* class its own *Update* method, which constantly checks if the level buttons are still in sync with their levels. You can then update a level button whenever its status is not correct. This is what that method looks like:

```

public override void Update(GameTime gameTime)
{
    base.Update(gameTime);
    foreach (LevelButton button in levelButtons)
    {
        if (button.Status != PenguinPairs.GetLevelStatus(button.LevelIndex))
            button.Status = PenguinPairs.GetLevelStatus(button.LevelIndex);
    }
}

```

You could also think of a different solution that immediately updates the correct level buttons as soon as the player finishes a level. This is a bit harder to program (you'll have to change several classes), but it's a nice exercise if you're up for a little challenge.

Functionally, the game is now completely finished. Almost there!

21.3 Adding Music and Sound Effects

To funk up the Penguin Pairs game,¹ we've prepared three sound effects and one piece of background music. Add these to the project (in a "Sounds" folder), and use the `AssetManager` class to play them at the appropriate times. Here are some guidelines:

- The background music (*snd_music*) should start playing when the game starts.
- The *snd_won* sound effect should play when the player finishes a level.
- The *snd_pair* sound effect should play when the player creates a pair of penguins that is *not* the final pair. (For the final pair, the player will already hear *snd_won*, and we don't want to play both sound effects at the same time.)
- The *snd_eat* sound effect should play when an animal dies, either by falling into the water or by ending up on a tile with a shark. In both cases, the `ApplyCurrentPosition` method of `MovableAnimal` is the right place to play the sound.

After you've added the music and sound, compile and run the game again, and check if everything works correctly. Note: now that the background music is here, you can also finally test the volume slider in the options menu.



CHECKPOINT: PenguinPairs5a

And that's it: the code of the game is now finished. Feel free to take a break by playing the game from start to finish. You've earned it!

21.4 Using Libraries and Namespaces

So far, you've created three different games in this book. Along the way, you've been building a reusable game engine. The code for that engine is now placed inside a folder named "Engine."

For the next game of this book, it would be nice if we could reuse this engine code *without* having to copy and paste the "Engine" folder to a different project. As we've already discussed a couple of times in this book, copying code around is a dangerous thing: if you ever discover a mistake somewhere or want to change the code for another reason, you'll have to make this change in many different places. That's also why good programmers use methods, classes, and inheritance to avoid duplicate code as much as possible.

But how can you avoid copying code *between different projects*? The answer is to create a *separate project* for the engine-related code. This project is not a game itself, but it's rather a **library** filled with code that your games can use. We've briefly talked about libraries before, way back in Chap. 3, to explain what the keyword **using** means. You've now reached the point where we return to this topic and apply it to our own projects.

¹We're sorry if "funk up" is no longer a cool term. We've been playing too much *DiscoWorld*.

In this section, you'll learn how to use a library for the code that you want to share across different projects. We'll also show how to use *namespaces* to structure your projects even further. The result can be found in the PenguinPairsFinal example project of this chapter.

You don't have to apply these changes to your *own* project if you don't want to. Instead, we'll just explain the structure of our PenguinPairsFinal project, and you're encouraged to take a look at that project while reading.

21.4.1 A Separate Library for the Engine

The Visual Studio solution for this chapter contains three projects: PenguinPairs5a, PenguinPairsFinal, and Engine. The PenguinPairsFinal project contains the same code as PenguinPairs5a, but we've left out the entire "Engine" folder. Instead, this project now uses the Engine project as a library. That way, the PenguinPairsFinal project itself only contains code that is really specific for the Penguin Pairs game.

The Engine project is a special type of Visual Studio project called a *class library*. If you compile this project, it will not result in an executable game that you can run. Instead, it will create a *.dll* file that other projects (such as PenguinPairsFinal) can use. In each game project that you're going to make, you can just refer to this same Engine library again, without having to copy and paste the code. The next game of this book (Tick Tick) will also use this library.

It's a bit tricky to create a library with the correct settings in Visual Studio, and it's not extremely interesting to discuss, so we won't explain it in this book. If you ever do this yourself and you run into problems, you can find a lot of information online.

What *is* useful to know is that you have to tell Visual Studio that projects refer to each other. In this case, Visual Studio should know that the PenguinPairsFinal project uses the code from the Engine project. That way, if you compile and run the PenguinPairsFinal project, Visual Studio will know that it needs to compile the Engine project first, and then use the resulting *.dll* file for compiling PenguinPairsFinal.

To add such a reference in Visual Studio, look at the Solution Explorer panel and right-click on the project (in this case PenguinPairsFinal) that should receive a reference to another project (in this case Engine). In the menu that appears, choose Add → Reference... and find the project that you want to refer to. Mark the checkbox next to "Engine", as shown in Fig. 21.1, and click OK. This applies the reference. (In the PenguinPairsFinal example project, we've already done this for you.)

21.4.2 Access Modifiers for Classes

Way back in Chap. 7, we talked about **access modifiers** such as **public** and **private**. You can use access modifiers for methods, properties, and member variables—but you can also apply them to entire *classes*!

The two most relevant access modifiers here are **internal** and **public**. A public class (or interface) can be used everywhere in your program, but an internal class can only be used within the same so-called assembly. By default, classes and interfaces in C# are internal, unless you explicitly write the word **public** in their header.

The Engine library that we've just made is a separate assembly. So, by default, all classes and interfaces in this library can only be used within the Engine library itself. We need to explicitly mark these classes and interfaces as **public** if we want to use them in other projects.

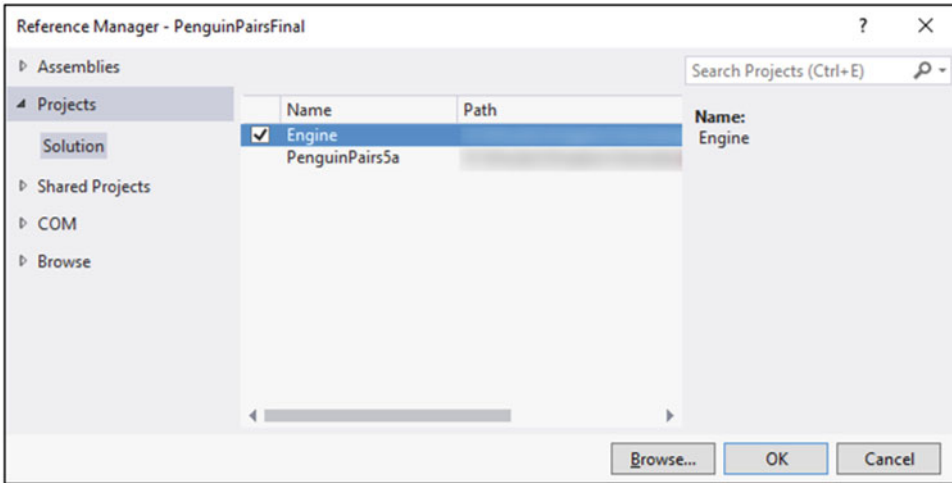


Fig. 21.1 In Visual Studio, you can let the PenguinPairsFinal project *refer* to the Engine project

If you browse through the Engine library in our final example project, you'll see that we have indeed added the keyword **public** to the headers of all classes and interfaces. This will allow us to use those classes and interfaces in other game projects.

Why Use Internal Classes? — Classes are internal by default, but that may sound a bit weird. What is the use of having classes that can only be used within a library and not outside it?

The main reason is that it allows developers of a library to decide which parts of the code are available to users, and to hide the “low-level” parts that users don't have to worry about. For example, suppose that you're developing a library for dealing with collisions in games. You could imagine that the goal of this library is to answer a few basic types of questions that users want to ask: Does one object collide with another object? What is the exact position where two objects collide? On the other hand, there will probably be a lot of low-level mathematical code that you *don't* want to show to your users.

The advantage of using internal classes for low-level code is that your library is less intimidating to use. By only exposing the things that are really important, the library becomes easier to understand for your users.

21.4.3 A Namespace for the Engine Classes

A final step for structuring the code even further is to put all elements of the Engine project inside a **namespace**. Up until now, we've always ignored the concept of namespaces, because our projects were not that complex yet. But now that the Penguin Pairs game has grown into a larger project with a separate reusable library, it's a good idea to start using namespaces as well.

In our Engine project, all classes and interfaces now have the following code around them:

```
namespace Engine { ... }
```

And in the “UI” folder of that project, we’ve even used a more specific namespace:

```
namespace Engine.UI { ... }
```

Now, whenever the PenguinPairsFinal project wants to use elements from this namespace, it has to explicitly say that they’re from Engine (or Engine.UI). As you may recall from Chap. 3, there are two ways to do this: you can either write “Engine.” in front of the class (or interface) name when you need it or you can add the text “**using** Engine;” at the top of the file. In our example project, we’ve chosen for the second option most of the time. For example, the *LevelButton.cs* file starts with the following two lines:

```
using Engine;
using Engine.UI;
```

because it needs the Engine namespace to be able to use classes such as *TextGameObject*, and it needs the Engine.UI namespace for the *Button* class.

Sometimes, we use the first option if the term “Engine.” is needed only once in a file. An example is the header of the *Animal* class:

```
abstract class Animal : Engine.SpriteGameObject
```

Do you notice how the Engine namespace is now comparable to namespaces that you’ve included before, such as *Microsoft.Xna.Framework*? In large projects, it’s very common to use many different namespaces, and (as a result) you’ll often use the word “**using**” for namespaces and libraries that you’ve written yourself.

21.5 What You Have Learned

In this chapter, you have learned:

- how to reset a level to its initial state and how to move between levels;
- how to create a separate library with classes that are used by another project;
- the difference between public and internal classes;
- how to use namespaces to structure your projects even further.

The Penguin Pairs game is now finished. As usual, in the exercises for this chapter, we’ve added some suggestions for extending the game.

In terms of code, Penguin Pairs is already quite a bit more complicated than the previous example game, Jewel Jam. You’ve probably noticed that the number of classes is now quite big, and we’re relying more and more on certain design choices in the program. For example, we have chosen to organize game objects in a tree structure, and we have designed a class for handling game states. Furthermore, we assume that game objects are responsible for handling their own input, updating themselves, and drawing themselves on the screen.

You may not agree with some (or any) of these design choices. At this point in the book, perhaps you’ve formed your own ideas about how game software should be designed. That is a good thing. We’ll say it once again: there are *many ways* to design a game program, and the design that we propose in this book is definitely not the single perfect option.

During game development, you can always evaluate your design, improve it, or even rewrite it completely. Don't be scared to give your code an overhaul every once in a while—after you've created a safety backup, of course. By trying out different approaches to solve a problem, you'll understand that problem better, and you'll become a better software developer as a result.

21.6 Exercises

1. *Classes and Access Modifiers*

Here are two questions about access modifiers for classes.

- a. What is the difference between an internal class and a public class?
- b. By default, a C# class is treated as if it has the access modifier **internal**. Can you think of a reason why this is the default access modifier for classes, instead of **public**?
- c. * In our Engine project, we can't mark any of the classes as **private** or **protected**. Why is this? By the way, it *is* possible to use these keywords for a class in some cases. Can you imagine how?

2. *Extending the Penguin Pairs Game*

Below are some suggestions for extending the Penguin Pairs game. As usual, we've sorted the suggestions by difficulty, based on how much they require your program to change. Again, you're completely free to think of your *own* extensions as well!

- a. (*) Currently, the level menu consists of a single page displaying a maximum of 15 levels. Extend the game so that multiple pages of levels are allowed. Add two buttons to the level menu screen to be able to navigate through the different pages.
- b. (*) Add a menu option that lets the player reset his/her progress. When the player clicks that button, the first level should get marked as "unlocked," and all other levels should get marked as "locked."
- c. (**) If you're a quick player, it's currently possible to move multiple animals at the same time. This is not really what the levels were designed for. Extend the game so that the player cannot make any moves as long as an animal is moving.
- d. (**) Allow levels to specify a maximum number of moves, so that the player can only click on arrows a limited number of times. If the player runs out of moves, he/she loses and should restart the level.
- e. (***) Add a "booster" tile to the game that changes the direction of an animal that moves over it. For example, if a penguin is moving upward and then enters a booster tile of the type "right," the penguin should change its direction so that it continues moving to the right. Add a few more levels to the game that use this tile.
- f. (****) Extend the game so that it also contains polar bears that can be moved by the player. Whenever a penguin or seal collides with a polar bear, the penguin or seal gets so scared that it immediately starts moving away from the bear. Design a few levels around this concept.
- g. (*****) Add "teleporter" objects to the game. Teleporters should always come in pairs. Whenever a penguin or seal enters one teleporter, it should disappear and then reappear in the *other* teleporter, and it should continue moving in the same direction that it had before. Make it possible to have *multiple* pairs of teleporters in one level. Think carefully about how you represent teleporters in your text files. How can you tell the game which teleporters are "connected" to each other?

Part V

Animation and Complexity

Introduction



In this final part of the book, you’re going to build a 2D platform game called *Tick Tick*, which contains animated characters, physics, and different levels. You’ll mostly learn about game-specific topics, such as:

- how to create animations by showing different sprites over time;
- how to add physics for a character that can move, jump, fall, and collide with other objects;
- how to create advanced intelligent behavior for enemies.

Next to this, you’ll also learn some final C# concepts that we think are important to understand. Most importantly, we’ll teach you about *exceptions*, which form a neat tool for handling unexpected errors in your code.

You will reuse a lot of elements from previous parts of the book. In particular, you’ll reuse the Engine project from the previous chapter. You’ll make some final changes to this engine to make it easier to reuse.

Tick Tick is by far the biggest and most complex game from this book. However, many components of the game are very similar to what you've seen before, and it wouldn't be very interesting to talk about everything all over again. Therefore, the setup of this final part of the book is a bit different: we'll start with a project that already does a lot of basic work, and then we'll extend it with interesting new features. We'll keep the text rather short, focusing on the concepts that are really new.

One of the main challenges of this part is to learn how to deal with a large codebase that already exists. Now that you understand the basics of programming, your main challenge as a programmer is to decide when to use what technique, without getting lost in the code you already have. This requires a lot of training—so consider the Tick Tick game a nice preview of what your future as a (game) programmer might hold!

About the Game: *Tick Tick* revolves around a slightly stressed out bomb that is going to explode within 30 s. This means that each level in the game should be finished within 30 s. Under the player's control, the bomb can walk and jump its way through a level. Each level contains a number of water drops. The bomb should collect all of these water drops *and* reach the finish panel in time. Throughout the game, we'll introduce several enemies and other hazards to make things more difficult for the player.

To play the final version of this game, open the solution belonging to Chap. 26, and compile and run the TickTickFinal project.

Chapter 22

Creating the Main Game Structure



In this chapter, we will first give an overview of the framework that you'll use for the Tick Tick game. This framework relies on a lot of classes that you've already written before. In fact, we'll build the game upon the Engine library that you developed earlier for Penguin Pairs. This means that we already have a basic design for a hierarchy of game objects, handling game states, and more.

From now on, each example project of this book will reuse the same Engine library, stored at a fixed location in our example folders. This version of *Engine.csproj* is a separate project *without its own solution around it*. Other solutions will include this project.

Next to introducing the starting point for Tick Tick, this chapter will also discuss how to deal with large classes in your code and how to handle unexpected errors in your game via *exceptions*. This is one of the final programming concepts that you'll learn in this book.

At the end of this chapter, you'll understand the initial code for Tick Tick. You can then start implementing the rest of the game by studying the next chapters.

22.1 Changes in the Engine Project

The Tick Tick game will have many similarities to Penguin Pairs. There will be a title screen that allows us to go to the level selection screen or to a help page. Each level will be read from a separate text file, and the player's progress is read from (and written to) a text file as well. Each level has a certain status: locked, unlocked, or solved. To keep things simple, we won't implement an options menu this time around, even though adding it would be very straightforward: we could simply reuse more code from Penguin Pairs.

The *Engine.csproj* project that we'll use from now on is an extended version of the Engine project from Penguin Pairs, with extra code for the parts that both games have in common. This makes the code even more reusable for future games.

In this section, we'll give a quick overview of the changes in *Engine.csproj* compared to the Penguin Pairs version. Go ahead and look for this project now, so that you can put the code and our explanation side by side.

22.1.1 Classes for Games with Levels

First of all, we've added an abstract class `ExtendedGameWithLevels` to the engine. You can find it in the `Levels` subfolder of the project. This class is a subclass of `ExtendedGame` that adds the structural features from `Penguin Pairs` that will now reoccur in `Tick Tick`:

- The `LoadProgress` method loads the player's progress from a text file. It fills a static list of `LevelStatus` values. (We've moved the `LevelStatus` enum to the Engine project as well.)
- There are static methods for getting and setting the status of a level, for marking a level as solved (and possibly marking the next level as unlocked), for saving the player's progress to a file, and for going to the next level.

To enable the last feature of this list, we have to promise `ExtendedGameWithLevels` that there will be a game state that is capable of loading a level. But because each game can have its own type of levels, the Engine cannot really predict what "loading a level" means: the details will be different for each game. To promise that loading levels is *possible* (without specifying the details yet), we've added an interface: `IPlayingState`. This interface contains two methods: `LoadLevel` (for loading a specific level) and `LevelCompleted` (for all the things that need to happen when the player completes a level). For convenience, we've given `ExtendedGameWithLevels` a method `GetPlayingState` that returns the game's playing state as an `IPlayingState` object. Each specific game (such as `Tick Tick`) will need to make sure that there really *is* a playing state that implements this interface.

The `LevelButton` class from `Penguin Pairs` has also moved to the Engine project. We've done this because the concept of "a button that represents a level with a status" is so general that it'll be useful in many games. We've marked its `label` variable as **protected** because its position and font will be different for each game (and some games may not even *need* it!).

22.1.2 Adding Depth to Game Objects

In `Penguin Pairs`, it was starting to become a bit annoying that sprites (and text) were always drawn in the order in which we added them to the game world. For example, because we created the "level completed" image before we could load the level itself, we always had to make the level invisible whenever we wanted to show that image.

As an alternative option, `MonoGame` also allows you to draw each sprite at a certain *depth*, via the last parameter of `spriteBatch.Draw`. So far, we've always ignored this option, using a depth of zero everywhere. In the new Engine project, all visual objects (such as `SpriteGameObject`, `TextGameObject`, `SpriteSheet`, and `Button`) have now received a new parameter `depth` in their constructors. This `depth` value is used when drawing the object, instead of a fixed depth of zero.

Next to giving the objects a `depth`, we also have to tell `MonoGame` that we want to draw sprites based on their `depth`. This can be done by changing the first parameter of the `spriteBatch.Begin` method, which you can find in our `ExtendedGame` class:

```
spriteBatch.Begin(SpriteSortMode.FrontToBack,
    null, null, null, null, spriteScale);
```

By changing `SpriteSortMode.Deferred` into `SpriteSortMode.FrontToBack`, `MonoGame` will draw the objects based on their `depth`, from lowest to highest.

The drawing `depth` in `MonoGame` always has to lie between 0 and 1. We recommend using a `depth` of 0 for background images, a `depth` of 1 for UI elements that are shown on top, and values in-

between for the remaining game objects. If two overlapping images have the exact same depth, it's a bit unpredictable what MonoGame will do, so be careful.

In the TickTick1 project (which we'll discuss soon), we've made it a bit easier to work with depths, by predefining a number of *constants* in the TickTick class:

```
public const float Depth_Background = 0; // for background images
public const float Depth_UIBackground = 0.9f; // for UI elements with text on top of them
public const float Depth_UIForeground = 1; // for UI elements in front
public const float Depth_LevelTiles = 0.5f; // for tiles in the level
public const float Depth_LevelObjects = 0.6f; // for all game objects except the player
public const float Depth_LevelPlayer = 0.7f; // for the player
```

We recommend using such constants everywhere, instead of manually chosen numbers. If you ever need a new depth value, consider adding it to this list of constants.

22.1.3 Other Changes

There are a few more changes in the Engine project that are worth talking about. First of all, the constructor of `SpriteGameObject` also accepts `null` as a sprite name. In that case, the object will not (yet) have a sprite when it is first created. This is useful if a subclass of `SpriteGameObject` wants to call the base constructor first and set the sprite later. To compensate for this choice, `SpriteGameObject` should now also take into account that its `sprite` member variable might be `null`.

Second, sprites can now be *mirrored*. The `SpriteSheet` class has a new property (`bool Mirror`) that says whether or not the sprite should be drawn in a mirrored way. In the `Draw` method, we use this property to draw the sprite in the desired way. One of the parameters of the `spriteBatch.Draw` method is a value of the `SpriteEffects` enum. So far, we've always been using the expression `SpriteEffects.None`, indicating that the sprite should not be transformed in any way. From now on, we use a value that depends on the `Mirror` property:

```
SpriteEffects spriteEffects = SpriteEffects.None;
if (Mirror)
    spriteEffects = SpriteEffects.FlipHorizontally;
spriteBatch.Draw(sprite, position, spriteRectangle, Color.White,
    0.0f, origin, 1.0f, spriteEffects, depth);
```

Mirroring sprites will be very useful in Tick Tick, because we'll have characters that can move to the left and to the right. Instead of requiring separate sprites for both directions, we can just use a single sprite and mirror it when necessary.

Third, we've changed the `LocalPosition` property of the `GameObject` class. It now has a protected member variable `localPosition` behind it:

```
public Vector2 LocalPosition
{
    get { return localPosition; }
    set { localPosition = value; }
}
protected Vector2 localPosition;
```

Why did we do this? Well, this is useful if an object ever wants to change just one coordinate (*x* or *y*) of its position. The following instruction is not allowed:

```
LocalPosition.X += 10;
```

because technically, the “get” part of the `LocalPosition` property works as a method. And because `Vector2` is a *struct* (and not a class), this property actually returns a sort of copy, instead of the original object. Changing the *x* or *y* component of that copy will not have any effect on the game object. The C# compiler recognizes this problem for you, and it forbids you to write something like this.

By adding a `localPosition` member variable, all subclasses of `GameObject` can now directly edit their position. The following instruction *is* allowed:

```
localPosition.X += 10;
```

because it manipulates the actual position of the object, and not a copy of it.

Fourth, for completeness, we’ve given the `InputHelper` class two extra methods for keyboard input: `KeyDown` (which checks if a certain key is currently being held down) and `KeyReleased` (which checks if the player has stopped pressing a certain key in the last frame). Finally, we’ve changed `TextGameObject` so that the text color can be changed by other objects during the game.

The Engine project contains a few more classes that we haven’t talked about yet. These classes are related to topics that we’ll treat in Chaps. 23 (animation) and 24 (game physics). Feel free to ignore these parts of the engine for now; we’ll discuss them further when the time is right.

22.2 Overview of the Initial Game

Now that you’ve seen the updated Engine project, let’s look at the first version of Tick Tick. This first version is capable of loading tile-based levels from files, but it does not yet contain a controllable bomb character, enemies, or other game logic.



CHECKPOINT: TickTick1

The TickTick1 example project of this chapter contains our starting point for this game. We’ll give you an overview of this project now.

To keep things simple, the TickTick1 project already includes *all assets* that you will need for the entire Tick Tick game. So, in this part of the book, we won’t tell you to include a new asset when you need it.

22.2.1 Main Classes

The main class of this game, `TickTick`, is a subclass of `ExtendedGameWithLevels`. Therefore, it already inherits most of the work that it needs to do. The only methods that it still needs to implement are `Main` (the starting point that every program needs) and `LoadContent` (for setting the size of the game world, loading specific game states, and playing background music). We’ve also created a custom constructor that makes the mouse pointer visible. Take a look at the `TickTick` class and check if you can understand everything.

This game has four game states: a title screen, a help screen, a level selection menu, and a playing state. Each state has a separate class in the `GameStates` folder. The first three game states are very similar to the ones of `Penguin Pairs`, so we won’t discuss them again here.

The `PlayingState` class is also quite similar to the `Penguin Pairs`: it contains a “quit” button, an overlay that will be shown when the player completes a level, a “game over” overlay, and a member

variable `Level level` that stores the level that is currently being played. The main difference to `Penguin Pairs` is that this class now implements the new `IPlayingState` interface, to promise that it indeed implements the `LoadLevel` and `LevelCompleted` methods. Please go through the `PlayingState` class now to verify this.

There is also a `Level` class again, which you can find in the file *Level.cs*. Just like before, a level is mostly a `GameObjectList` with many child objects, and it has member variables with extra references to the objects we'll need more often. Specifically, a level stores extra references to its tiles (in a 2D array), its water drops (in a list), and its goal object. Let's look at these objects one by one.

22.2.2 The Tile Class

Just like in `Penguin Pairs`, the levels of `Tick Tick` will consist of dynamic objects (such as water drops and enemies) and objects that never change. The non-changing parts of a level will be represented by *tiles*. Each level contains a rectangular grid of tiles. Each cell of this grid contains an instance of the `Tile` class. This `Tile` class is comparable to the one we used in `Penguin Pairs`, but it has a few differences. Each tile has a certain type:

```
public enum Type { Empty, Wall, Platform };
Type type;
```

An empty tile is invisible; empty tiles do not have a sprite. A wall tile is a block on which the player can stand. A platform tile is a special kind of block: the player can stand on it *and* jump onto it from below. We'll implement this behavior later.

A new feature is that walls and platforms have a certain type of *surface*:

```
public enum SurfaceType { Normal, Hot, Ice };
SurfaceType surface;
```

A normal surface doesn't do anything special. A hot surface will make the game's timer decrease more quickly. An ice surface will be slippery, making the character more difficult to control. Again, we'll implement these special features in later chapters.

The constructor of `Tile` takes a tile type and a surface type as its parameters. Based on these parameters, it will load the correct sprite, just like in `Penguin Pairs`. Note: technically, this code will also give *empty* tiles a surface type, which doesn't really make sense. For simplicity, we'll just ignore the surface type of empty tiles.

22.2.3 The WaterDrop Class

The only *dynamic* game object that we've already implemented is the *water drop*. In the next chapters, we'll add more game objects, such as the player and a variety of enemies.

The goal of each level in `Tick Tick` is to collect all water drops. Each water drop is represented by an instance of the `WaterDrop` class. This class is a `SpriteGameObject` subclass, but we've added a visual effect that makes a water drop slowly bounce up and down. `WaterDrop` has a custom `Update` method that uses the `Math.Sin` method to smoothly change the drop's *y*-coordinate. To make sure that not all water drops move in the exact same way, the bouncing behavior depends on a drop's *x*-coordinate as well. Have a look at the `WaterDrop` class if you're interested in the details.

22.2.4 The Goal Object

Each level also contains a *goal* object at a certain position. After having collected all water drops, the player should go to that object before the time runs out. Because this goal object doesn't do anything special, we've chosen not to create a separate class for it. Instead, this object is just a `SpriteGameObject`. The `Level` class stores it in an extra member variable for easy access, because we'll want to check for collisions between the player and this object.

22.2.5 Level Files

Again, levels will be stored in text files. Each level file starts with a one-sentence description that will be shown on the screen. Apart from that, a file contains a grid of characters that represent the tiles of the level, where each empty tile can possibly contain a dynamic object as well (such as a water drop).

The following symbols will be used to describe the *tile* part of a grid cell:

- '.' represents an empty tile;
- '#' represents a wall tile;
- '-' represents a platform tile that the player can jump through;
- 'H' and 'h' represent a hot wall and a hot platform, respectively;
- 'I' and 'i' represent an ice wall and an ice platform, respectively.

These symbols are already supported in the `TickTick1` project. We're already supporting a few other level elements as well:

- 'W' represents a water drop;
- '1' indicates the starting position for the player;
- 'X' indicates the position where the goal object will appear.

For example, the file for the first level looks like this:

```
Pick up all the water drops and reach the exit in time.
.....
.....X..
.....#####
.....
WWW...WWW.....
---...####.....
.....
WWW.....
###.....WWWWW...
.....#####
...WWW.....
...###.....
.....
.1.....W.W.W.W.
#####
```

The other levels of the game contain *enemies*, which we'll add in a later chapter. These enemies are represented in a text file by the following symbols:

- 'R' represents a rocket enemy that moves through the screen;
- 'T' represents a turtle enemy that can be jumped on;
- 'S' represents an electric enemy named "Sparky" that drops down;
- 'A', 'B', and 'C' represent three variants of a flame enemy that walks around.

For now, we'll ignore this last set of symbols, because we don't worry about the game's enemies yet. And just like in Penguin Pairs, all unsupported symbols will be treated as an empty tile. So, at the moment, tiles with enemies will simply be treated as empty tiles.

22.2.6 Loading a Level

The code for loading a level from a text file is (again) very similar to the code from Penguin Pairs. You can find this code in the *LevelLoading.cs* file. We open a file, read the level's description, and then read the list of grid rows. There's an *AddPlayingField* helper method that initializes the 2D grid of tiles and then goes through all symbols in the file. For each symbol, we call the *AddTile* method.

The *AddTile* method first loads the non-changing part of a grid cell. This becomes a *Tile* object that gets added to the game world at the correct position, as well as to the 2D grid. We've added a helper method *CharToStaticTile* that uses a **switch** instruction to convert symbols to the correct tile type. The **default** case creates an empty tile.

After that, the *AddTile* method loads the remaining part of a grid cell: the part that cannot be represented by a *Tile* object. For now, we support the symbols '1', 'X', and 'W', with the meanings explained before. The helper methods *LoadWaterDrop* and *LoadGoal*, respectively, add a water drop or a goal object to the level. The helper method *LoadCharacter* doesn't do anything yet; this method should create a game object for the bomb character, but we don't have a class for that yet.

All of the methods in the *LevelLoading.cs* file are **private**. This means that only methods inside the *Level* class can access these methods. We did this because only the *Level* class itself really needs these methods, so it doesn't make sense to make them publicly accessible. All of these methods will (indirectly) be called by the *Level* constructor. If we allowed these methods to be called from everywhere, then programmers could accidentally mess up the information stored in a level. Making the methods private makes sure that the code is nicely isolated from the rest of the program.

This concludes our discussion of the *TickTick1* project. Compile and run this project to see the game in action. Now's a good moment to change the *levels_status.txt* file so that each level is marked as unlocked. This will allow you to look at all levels. (Of course, for the final version of *Tick Tick*, you'll have to revert this so that the player needs to unlock the levels by playing the game.)

22.3 Dealing with Large Classes

Given everything that we want to do in the game, the *Level* class is most likely going to be quite large. We've already written code for loading a level from a file. In later chapters, we'll add even more code for the *game loop*: the special behavior of a level that needs to be performed in each frame.

This large amount of code can make the class difficult to read for other programmers (and for yourself). However, splitting the code into multiple classes (just to increase readability) is not really an option, because that's not what classes are meant for.

In this section, we'll describe a few ways to deal with large classes more easily. These are really just tricks and not key programming concepts, but you might come across them in your career as a programmer.

22.3.1 Partial Classes

One option is to divide a class over multiple files. We've done this in the TickTick1 project: the files *Level.cs* and *LevelLoading.cs* both describe a certain part of the *Level* class. So, the full class definition is given by these two files combined.

A description of a *part* of a class (instead of the entire class) is called a **partial class**. In C#, you can say that a class description is partial by adding the keyword **partial** to the class header. You have to do this in *all* parts of the class. For example, the class headers in *Level.cs* and *LevelLoading.cs* both look like this:

```
partial class Level : GameObjectList
```

Note that you also need to repeat any inheritance information in every part. By contrast, each method, property, or member variable should only appear in *one* of the parts. For instance, *Level.cs* declares the member variable `Tile[,] tiles` while *LevelLoading.cs* does not.

The **partial** keyword does nothing besides allowing the code to be spread over different source files. The compiler will combine all these source files into a single *Level* class when it compiles the files.

The downside of using partial classes is that it's not immediately clear which elements can be found in which parts. If your class is divided into, say, 10 parts, then it'll probably take some time for another programmer to find a certain method or member variable.

22.3.2 Regions

Another option is to use a single file but to clearly divide it into separate "sections" that can be opened and closed in Visual Studio. Such a section is called a **region**. In Visual Studio, you can create a region by surrounding your code by the following two lines:

```
#region Your region name  
...  
#endregion
```

Of course, Your region name can be replaced by any description that you think is useful.

In Visual Studio, if you press the "-" icon next to the first line of the region, then all code inside that region will (temporarily) be hidden from your view. This is a nice way to show and hide different pieces of code within the same file.

By the way, regions are not really part of the "standard" C# language. They're actually a Visual Studio-specific trick for helping you organize your code.

In our opinion, regions are nicer than partial classes because they're a bit less "extreme." We sometimes catch ourselves creating a region even if the class file really isn't that long yet. It's just an extra way to make your life as a programmer a bit easier.

Watch out: if a class becomes extremely long (such as thousands of lines), then it's often a sign that your code can be improved in *other* ways. While it's tempting to use regions and partial classes just to hide the fact that your class is very large, we strongly advise against that. It's much more important to write nicely structured programs using methods, classes, inheritance, comments, and all other programming concepts from this book.



Quick Reference: Dealing with Large Classes

If you have a very long class and you want to make the code more readable, there are (at least) two things you can use:

1. *Regions*. Within a single file, write the following lines around a group of code elements:

```
#region Your region name
...
#endregion
```

This allows you to quickly show and hide parts of a file in Visual Studio. Note: this is specific for Visual Studio, and it's not officially part of the C# language.

2. *Partial classes*. Divide the class over multiple files, and use the word **partial** in the class header of all of these files:

```
partial class MyClass { ... } // in file 1
partial class MyClass { ... } // in file 2, and so on
```

22.4 Exceptions: Dealing with Unexpected Errors

Looking back at level loading, there's one important thing that we haven't talked about yet: *What should happen if something goes wrong while loading a level?* For example, what if the file with the given name cannot be found? Or what if the contents of the file don't match the file format that we've made up? The compiler cannot detect those kinds of problems beforehand, because files aren't loaded until the game really starts. Luckily, C# has a very nice way of dealing with such "unexpected errors": **exceptions**.

An exception is a special "message" created by your program, which says that something has gone wrong. In programming terminology, an exception gets *thrown* by the program if something goes wrong. If an exception occurs and you don't do anything special, then your program will crash.

You may have seen exceptions (and therefore crashes) in your own code a few times already.¹ For example, exceptions occur when you try to access an array element that doesn't exist (such as the element at index `-1`), when you try to convert non-numeric text to a number (using `int.Parse` or `double.Parse`), or when you try to access an object via a variable that is still `null`. These kinds of exceptions are often caused by programming errors. In those cases, you'll usually want to fix your code and recompile the program.

However, some exceptions are beyond your control as a programmer. File I/O is a nice example of this: it could happen that a file doesn't exist or that its contents are in the wrong format. Another example is network communication. If you've written an online game that connects to a server (e.g., to load and save high scores), it's possible that the server is unavailable, or that the player's Internet connection is temporarily down. In those cases, you can't easily "fix the code," and your program will have to deal with the errors in a nice way that doesn't harm the player's experience.

¹Or *many* times—that's nothing to be ashamed of!

22.4.1 Dealing with Exceptions: *try* and *catch*

As said, the easiest way to deal with an exception is to do nothing. Your program will then crash and the player will have to restart your game. Obviously, this isn't the best way to treat your audience.

A better option is to write special code that gets executed when the exception occurs. This is called *catching* the exception. In C#, the first step is to indicate that exceptions *can occur* in a certain part of the code. You can do this by placing the code inside a **try** block, like this:

```
try
{
    // your code here
}
```

If no exceptions occur inside the **try** block, then the code will behave normally. If an exception *does* occur, then you need to indicate what the program should do instead. This is done via a **catch** block, which follows immediately after the **try** block:

```
catch (Exception e)
{
    // your alternative code here
}
```

Whenever the program executes the code inside the **try** block and an exception occurs, the program will immediately jump to the **catch** block, without finishing the rest of the **try** block. Inside the **catch** block, "Exception e" acts almost like a method parameter: it's a local object with details about the specific exception that has occurred. Here, Exception is a class type defined somewhere in the C# language, and e is just a name that you could also replace by something else. You can use this parameter to (for example) write information about the exception to the console.

For **try** and **catch**, the curly braces are mandatory: you're not allowed to leave them out, even if a block consists of only one instruction. This is a bit illogical, given that the braces may be omitted in many other instructions such as **if** or **while**.

An example of a method that can throw exceptions is `int.Parse`. If the string that you give to it cannot be translated to an integer, an exception occurs. Let's say that `txt` is a variable of type **string**. The following code uses **try** and **catch** to call `int.Parse(txt)` in a safe way:

```
try
{
    int n = int.Parse(txt);
    // do something with n
}
catch (Exception e)
{
    Console.WriteLine(txt + "is not a number");
}
```

In this example, if `txt` can be turned into an **int** without problems, then the code inside the **catch** block will just be ignored. But if the parsing goes wrong, the program will skip the rest of the **try** block and jump into the **catch** block, which writes a message to the console.

22.4.2 Multiple Types of Exceptions

The Exception class is a general class that can represent any type of exception. However, C# has defined many *subclasses* of Exception that represent more specific errors. Some examples that you

may have seen already are `IndexOutOfRangeException`, `NullReferenceException`, `ClassCastException`, and `StackOverflowException`. Based on their names, you can probably guess what most of these exceptions mean. These subclasses all have their own special properties, member variables, or methods, and you can use those to obtain more information about what went wrong.

With **try** and **catch**, it's possible to treat different exceptions in different ways. To do that, you can write multiple **catch** blocks in a row and replace the term `Exception e` by a more specific exception type in each block. For example:

```
try { /* some code */ }
catch (NullReferenceException e) { /* Something was null! */ }
catch (ClassCastException e) { /* Casting went wrong! */ }
// etc.
```

The advantage of doing this is that you'll know more details about what went wrong, so you can respond to the exception in a more adequate way. Our advice is to always catch exceptions as specifically as possible. As a final safety net, you can always end with a general block:

```
catch (Exception e) {...}
```

to catch all remaining exceptions that you may have forgotten. This is kind of comparable to the “default” case of a **switch** block. The program will always jump into the first **catch** block that matches the current exception type.



Quick Reference: try and catch

The following code fragment:

```
try { ... }
catch (Exception e) { ... }
```

will first try to execute all code inside the **try** block. If an *exception* occurs somewhere in that block, the program will jump to the **catch** block and execute the code inside that block. There, the variable `e` will contain information about what went wrong.

You can use multiple **catch** blocks to deal with multiple types of exceptions.

22.4.3 Throwing Your Own Exceptions

But exceptions are not only things that your program can throw “by accident.” You can also create exceptions of your own! This may sound weird: why would you generate your own errors on purpose? Well, once you get the hang of using **try** and **catch** in the proper places, you'll find that exceptions are actually a very powerful programming tool.

To throw your own exceptions, you can use the **throw** instruction. This instruction consists of the keyword **throw** followed by an expression of type `Exception` (or any of its subclasses). For example, the following instruction:

```
throw new Exception("Something went wrong!");
```

creates a new `Exception` object and throws it. You could also throw an `Exception` object that already exists. For example, inside a **catch** block, you could choose to take the current exception and pass it on to the rest of the program:

```
...
catch (Exception e)
{
    ...
    throw e;
}
```

You're also free to create your own subclasses of the `Exception` class. This is useful if your game has very specific types of exceptions that C# itself doesn't know about. For instance, you could create a `InvalidLevelException` class that you can use when a level file has an incorrect format. Let's imagine that the first line of a level file should be an integer. You can then use **try** and **catch** to check if the first line can be parsed to an `int`. If that fails, then the level file is apparently incorrect, and you can throw your own `InvalidLevelException` with more information about the error:

```
...
string line = file.ReadLine();
try
{
    int n = int.Parse(line);
    ...
}
catch (Exception e)
{
    throw new InvalidLevelException("This level file does not start with a number.");
}
```

22.4.4 When to Use Exceptions?

This discussion of exceptions might be a bit overwhelming. At this point, the question “why/when should I use exceptions?” is a very logical one to ask.

As a main rule of thumb, an exception should always be what its name suggests: an *exceptional* case that shouldn't occur under normal circumstances. So, whether or not you should use exceptions depends on how “strange” it is for certain errors to occur.

In the first place, **try** and **catch** are useful for errors that are beyond your control, such as an invalid filename or a missing network connection. Also, if you're calling a method that is known to throw exceptions in certain cases (such as `int.Parse`), it's a good idea to use **try** and **catch**, unless you're completely sure that the exception cannot occur.

On the other hand, you should never “abuse” the **try** and **catch** keywords to deal with simple programming mistakes. For example, imagine a **for** instruction that loops over an array, but that always does one iteration too many:

```
for (int i=0; i <= myArray.Length; i++)
    ...
```

This will trigger an exception in the last iteration. (Just to be clear, the mistake is that we're using “<=” instead of “<” here.) To prevent the program from crashing, you could surround this code by **try** and **catch** instructions, but it's better to simply fix the error in the code itself.

Apart from these obvious cases, exception handling is partly a matter of taste. For instance, let's say that `myVariable` is a variable that could be `null` at some point. The following piece of code:

```
if (myVariable != null) { /* do something with myVariable */ }
else { /* do something else */ }
```

is pretty much similar to the following:

```
try { /* do something with myVariable */ }
catch (NullReferenceException e) { /* do something else */ }
```

Which of the two is better? That depends on how “exceptional” it is for `myVariable` to be `null`. For example, the `PlayingState` class in `Tick Tick` has a member variable `Level level` that is `null` until we load our first level. In this case, it's pretty normal for `level` to have the value `null` when the game has just started. That's why the `PlayingState` class uses `if` instructions (in several places) to check if a level has already been loaded. This is nicer than “abusing” the exception system of C# to handle a rather common case.

Long story short: exceptions are mostly meant for truly special cases that are beyond your control as a programmer. File I/O and network-related tasks are great candidates for this. In other cases, you'll have to think carefully about how “exceptional” a situation really is.

In the `Engine` and `TickTick1` projects, we haven't added any exception handling, to keep things simple. However, it's definitely possible to add `try` and `catch` instructions in several places. One of the exercises for this chapter invites you to think about that.

22.4.5 Exceptions and Casting: The Keyword `as`

One type of operation that can trigger an exception is a *casting* operation. Remember that a cast converts an expression to a different type. For example, if `myFloat` contains a value of type `float`, then the expression `(int)myFloat` will contain that value converted to an `int`.

We've said before that a cast is not always possible. For instance, you're not allowed to directly cast a string to an integer. Also, in our game engine, it's not possible to cast a `SpriteGameObject` to a `TextGameObject`, because they are different things. On the other hand, you *are* allowed a `SpriteGameObject` to a `GameObject`, because `GameObject` is the superclass of `SpriteGameObject`.

If you try to do an “illegal” cast during your game, the program will throw an *exception* that can crash your game. So, if you're not sure whether a cast will work, it may be a good idea to use the `try` and `catch` keywords you've just learned.

C# actually offers a second way to cast objects to another type. To convert an expression `myExpression` to the nullable type `MyType`, the “old-fashioned” way is to cast it as follows:

```
(MyType)myExpression
```

As said, this will cause an exception if the cast is not possible. The alternative way is to use the keyword `as`:

```
myExpression as MyType
```

This will do the following:

- If `myExpression` can indeed be converted to the type `MyType`, then this code does exactly the same as an ordinary cast.

- If the cast is not possible, then the result of this expression will be **null**, and there will *not* be any exception.

So, it's sometimes handy to cast using the keyword **as** (and not in the old-fashioned way with brackets), because it won't trigger an exception if the cast fails. However, you should then be aware that the result of the cast can be **null**. (If you don't deal with that possibility in the rest of your code, you might get *other* exceptions again!)

This second way *only* works for types that can refer to the value **null**, such as class types. It doesn't work for primitive types such as **int** and **float**. In other words, writing "myExpression **as int**" is not valid, because an **int** can never be **null**. (Types that support the value **null** are also called **nullable types**.)

22.5 What You Have Learned

In this chapter, you have learned:

- what the general structure of the Tick Tick program looks like;
- how to use regions or partial classes to make large classes more readable;
- how to deal with exceptions;
- how to use the keyword **as** for casting, as an alternative to the bracket notation.

22.6 Exercises

1. *Exceptions*

What is an *exception* in a C# program? Name a few scenarios in which exceptions are useful.

2. *Exceptions and File Reading*

This exercise starts with a simple file I/O method and then gradually lets you add exception handling.

- Write a method `ReadSum` that reads a file with a given filename. Each line of this file should contain an integer. The method should calculate and return the sum of all these integers. Give an example of a piece of code that calls `ReadSum` and does something with the result. (You've written a similar method in the exercises from Chap. 19, but now each number starts on a new line.)
- Improve the method so that when the given file does not exist, a message is printed to the console and a value of 0 is returned.
- Improve the method so that it ignores all lines for which the `int.Parse` method fails.
- Now change the method so that it throws a custom `InvalidLineException` as soon as there's an invalid line. (You may assume that the `InvalidLineException` class already exists and that it's a subclass of `Exception`.)
- Look back at your example usage of `ReadSum` from part (a). How should you change this code so that it handles your new `InvalidLineException`?
- Which version of `ReadSum` is your favorite? To what extent do you think exception handling is useful here?

3. * *Using the New Engine with Penguin Pairs*

With our new version of the Engine library, we can also improve the code of Penguin Pairs (the game from the previous chapters).

Update the PenguinPairsFinal project so that it uses the new version of Engine. Note that the PenguinPairs class should inherit from ExtendedGameWithLevels, and the PlayingState class should implement the IPlayingState interface. Are there more parts of the code that you can clean up now?

4. *Exceptions in Tick Tick*

Can you identify some places in the Engine and TickTick1 projects where exception handling could be useful? Give at least three examples, and explain why exception handling would make sense there.

Chapter 23

Animated Game Objects



In this chapter, you'll learn how to add **animations** to your game objects. Technically, animation is simply showing a series of (slightly) different images one after the other. This tricks the human brain into thinking that it sees smoothly changing objects.

This ties in nicely with the *game loop*, which draws the game world over and over again at 60 frames per second. So far, you've already created the illusion of *moving* objects by drawing sprites at different positions in each frame. In a very similar way, you could also *change the sprite* of an object over time. For example, Fig. 23.1 shows a sequence of sprites where each sprite is slightly different from the previous one. If we show these images one after the other, at just the right speed, we'll create the illusion that this character is moving its feet in a walking motion.

The goal of this chapter is to allow game objects to change their sprite over time. We'll first discuss a number of new classes in the Engine library that make this possible. Next, we'll use those classes in the Tick Tick game to create an animated bomb character. At the end of this chapter, you'll have extended the game with a character that can walk from left to right, based on which arrow key the player is pressing. The character will display different animations based on what it's currently doing.

23.1 A Class for Animations

First, let's discuss the Animation class that can be found in our Engine project. An instance of this class represents a single animation that can be played at a certain speed. Listing 23.1 shows the full Animation class; we'll talk about the details of this class now.

As you can see in the example from Fig. 23.1, it makes sense to place all frames of an animation inside a single image file. In our engine, each animation is actually also a *sprite sheet*, with one sprite for each frame of the animation. Therefore, the Animation class inherits from the SpriteSheet class that you've already made for the Penguin Pairs game. The *sheet index* of an Animation instance will indicate which frame we're currently drawing.



Fig. 23.1 A sequence of images representing a walking motion

Listing 23.1 The Animation class

```

1  using System;
2  using Microsoft.Xna.Framework;
3
4  namespace Engine
5  {
6      public class Animation : SpriteSheet
7      {
8          // Indicates how long (in seconds) each frame of the animation is shown.
9          public float TimePerFrame { get; protected set; }
10
11         // Whether or not the animation should restart when the last frame has passed.
12         public bool IsLooping { get; protected set; }
13
14         // The total number of frames in this animation.
15         public int NumberOfFrames { get { return NumberOfSheetElements; } }
16
17         // Whether or not the animation has finished playing.
18         public bool AnimationEnded
19         {
20             get { return !IsLooping && SheetIndex >= NumberOfFrames - 1; }
21         }
22
23         /// The time (in seconds) that has passed since the last frame change.
24         float time;
25
26         public Animation(string assetname, float depth,
27             bool looping, float timePerFrame) : base(assetname, depth)
28         {
29             IsLooping = looping;
30             TimePerFrame = timePerFrame;
31         }
32
33         public void Play()
34         {
35             SheetIndex = 0;
36             time = 0.0f;
37         }
38
39         public void Update(GameTime gameTime)
40         {
41             time += (float)gameTime.ElapsedGameTime.TotalSeconds;
42
43             // if enough time has passed, go to the next frame
44             while (time > TimePerFrame)
45             {
46                 time -= TimePerFrame;
47
48                 if (IsLooping) // go to the next frame, or loop around
49                     SheetIndex = (SheetIndex + 1) % NumberOfSheetElements;
50                 else // go to the next frame if it exists
51                     SheetIndex = Math.Min(SheetIndex + 1, NumberOfSheetElements - 1);
52             }
53         }
54     }
55 }

```


23.1.1 New Member Variables and Properties

Compared to a `SpriteSheet`, an `Animation` object has one main extra task: automatically changing its own sheet index after enough time has passed. To make this possible, we've added a property that says how many seconds each frame should be shown:

```
public float TimePerFrame { get; private set; }
```

We've also added the option to *loop* an animation. A looping animation will automatically return to the first frame after it has finished playing, so that the same animation will be shown over and over again. For example, if we keep repeating the animation from Fig. 23.1, it will look like the character is walking continuously. The sprite sheet contains just one “cycle” of the looping animation. Not *all* animations should be looped, though. For instance, Tick Tick will contain an animation for when the bomb character dies, and that animation should be played only once.¹

Thus, we want to be able to *choose* whether or not an animation should loop. There's a property that represents this choice:

```
public bool IsLooping { get; protected set; }
```

Furthermore, there's a read-only property that returns whether or not the animation has ended. This is the case when the animation has reached its last frame and *doesn't* loop.

The constructor of `Animation` has four parameters: the two parameters that `SpriteSheet` already had plus two extra parameters that will determine the values for `IsLooping` and `TimePerFrame`.

23.1.2 Playing an Animation

To let an animation play itself, the `Update` method should do something to automatically go to the next frame when enough time has passed. For this, we've added one more member variable, `float time`, which counts how much time has passed since the last time we went to a different frame.

We've added a `Play` method that should be called when we want to *start* playing the animation. This method sets the `time` variable to 0 s, and it sets the `SheetIndex` property to the desired index at which the animation starts. Usually, this index will be zero (so that the animation starts at the first frame), but we'll also allow animations to start with *another* frame. For example, the bomb character of Tick Tick has an animation for jumping, but it can also be used as an animation for *falling* if we skip the first half.

The most interesting method is `Update`. First of all, note that the header of this method does not contain the word **override**. This is because the parent class (`SpriteSheet`) does not have an `Update` method. So, for the `Animation` class, we have to write an `Update` method “from scratch.” For convenience, we've given it exactly the same parameter as all other `Update` methods that you've already seen.

The `Update` method first increases the `time` variable with the amount of time that has passed in this frame:

```
time += (float)gameTime.ElapsedGameTime.TotalSeconds;
```

You've used `gameTime.ElapsedGameTime.TotalSeconds` before, to give game objects a speed in pixels per second (instead of pixels per *frame*). We're using the same property here again, because it indicates exactly how much time has passed in the last frame.

¹Dying once is already bad enough!

After increasing the timer, we check if enough time has passed to trigger a frame change. If so, we go to the next frame by incrementing the `SheetIndex`. We should then also decrease the time variable as follows:

```
time -= TimePerFrame;
```

This will make sure that the *next* frame change will happen in exactly `timePerFrame` seconds. For example, imagine that `TimePerFrame` was set to 0.1 s (meaning that each frame should be shown for that amount of time). Now imagine that the value of time variable has become at least 0.1 for the first time. This means that at least 0.1 s have passed since the animation started, so we're ready to show the next frame now. By subtracting 0.1 from time, we make sure that this next frame will be shown for another 0.1 s, until time is (again) large enough to trigger another change.

There are some special cases when we've reached the end of the animation. If the animation is looping, we should return to the first frame. Otherwise, we should actually *not* increase the sheet index and stay at the final frame. The following code does this:

```
if (IsLooping)
    SheetIndex = (SheetIndex + 1) % NumberOfFrames;
else
    SheetIndex = Math.Min(SheetIndex + 1, NumberOfFrames - 1);
```

but there are several other ways to achieve the same effect. Can you think of an alternative for this code?

Finally, note that our `Update` method uses a **while** loop instead of an **if** instruction. This is because the animation frame might need to increase *multiple times* within a single frame of the game loop. This won't occur very often (it can only happen if `TimePerFrame` is smaller than the framerate of the game), but we've written it like this just for completeness. Usually, the loop will just have a single iteration.

23.2 Adding Animations to Game Objects

The `Animation` class provides the basis for playing animations. In this section, we'll use that class for a new kind of game object: the *animated* game object, which may store a number of different animations and play one of them at a time. For example, the bomb character in *Tick Tick* will have animations for walking, jumping, dying,² and more. An animated game object should be able to load and store these different animations and play a particular animation when the game asks for it.

The `AnimatedGameObject` class is a subclass of `SpriteGameObject`, with the extra feature that it can store multiple `Animation` objects at the same time. One of these animations will be set as the current sprite, via the `sprite` member variable of `SpriteGameObject`. This is allowed: because `Animation` is a subclass of `SpriteSheet`, the `sprite` variable can actually point to an `Animation` object as well.

Listing 23.2 shows the full `AnimatedGameObject` class. Let's discuss it in some more detail. First of all, note that the class does not have its own `Draw` method: the parent class already does the drawing for us, so we don't need a custom version of that method.

²Fun fact: "animation" literally means "bringing to life," so a death animation is pretty ironic.

Listing 23.2 The AnimatedGameObject class

```

1  using System.Collections.Generic;
2  using Microsoft.Xna.Framework;
3
4  namespace Engine
5  {
6      /// <summary>
7      /// A class that can represent a game object with several animated sprites.
8      /// </summary>
9      public class AnimatedGameObject : SpriteGameObject
10     {
11         Dictionary<string, Animation> animations;
12
13         public AnimatedGameObject(float depth) : base(null, depth)
14         {
15             animations = new Dictionary<string, Animation>();
16         }
17
18         public void LoadAnimation(string assetName, string id,
19             bool looping, float frameTime)
20         {
21             Animation anim = new Animation(assetName, depth, looping, frameTime);
22             animations[id] = anim;
23         }
24
25         public void PlayAnimation(string id, bool forceRestart=false, int startSheetIndex=0)
26         {
27             // if the animation is already playing, do nothing
28             if (!forceRestart && sprite == animations[id])
29                 return;
30
31             animations[id].Play(startSheetIndex);
32             sprite = animations[id];
33         }
34
35         public override void Update(GameTime gameTime)
36         {
37             base.Update(gameTime);
38
39             if (sprite != null)
40                 ((Animation)sprite).Update(gameTime);
41         }
42     }
43 }

```

23.2.1 Managing Multiple Animations

To store different animations inside an `AnimatedGameObject`, we use a member variable of type `Dictionary<string, Animation>`. Just like with our dictionary of game states, this will allow us to easily select a specific animation to play. The constructor of `AnimatedGameObject` initializes an empty dictionary.

The `LoadAnimation` method loads a new `Animation` object with the given image name and other properties. It then stores that animation in the dictionary under the given name.

The `PlayAnimation` method starts playing a previously loaded animation. The expression `animations[id]` returns the `Animation` object with the name `id`, assuming that this animation is indeed in the dictionary. (If that's not the case, then the game will crash, unless you add extra code for exception handling.)

`PlayAnimation` first checks if the chosen animation is already playing right now. If so, nothing happens, unless we're explicitly saying that the animation should always restart (via the `forceRestart` parameter). Next, we call the `Play` method for that animation (which will make it start playing), and we set it as this object's new `sprite`.

23.2.2 Updating the Current Animation

The final step is to make sure that the object's current animation (stored in the `sprite` property) actually updates itself, so that the `sprite` will change over time. The `SpriteGameObject` class doesn't do any updating, because that was never necessary up until now. So, the `AnimatedGameObject` class needs some extra code to make this possible.

This code can be found in the `Update` method. This method first calls `base.Update` to make sure that all standard updating behavior will always happen. Next, it checks if the current animation (stored in `sprite`) is not `null`. If that's true, then we should call the `Update` method of the current animation. But there's one catch: `sprite` is a variable of type `SpriteSheet`, and a `sprite sheet` cannot update itself. To be able to call the `Update` method, we explicitly have to convert `sprite` to an `Animation` object here, via a `cast`.

23.3 Adding the Player Character to Tick Tick

Let's use these new engine classes to create the first animated character of Tick Tick: the bomb character that can be controlled by the player. Add a `Player` class for this, in the `LevelObjects` folder. For now, we'll start with a simple version of the character that can only move horizontally or stand still, depending on keyboard input. In later chapters, you'll extend the `Player` class with much more behavior.

23.3.1 Overview of the Player Class

Listing 23.3 The first version of the `Player` class

```

1 using Engine;
2 using Microsoft.Xna.Framework;
3 using Microsoft.Xna.Framework.Input;
4
5 class Player : AnimatedGameObject
6 {
7     const float walkingSpeed = 400; // Standard walking speed, in pixels per second.
8     bool facingLeft; // Whether or not the character is currently looking to the left.
9
10    public Player() : base(TickTick.Depth_LevelPlayer)
11    {
12        // load all animations
13        LoadAnimation("Sprites/LevelObjects/Player/spr.idle", "idle", true, 0.1f);
14        LoadAnimation("Sprites/LevelObjects/Player/spr.run@13", "run", true, 0.04f);
15        LoadAnimation("Sprites/LevelObjects/Player/spr.jump@14", "jump", false, 0.08f);
16        LoadAnimation("Sprites/LevelObjects/Player/spr.celebrate@14", "celebrate", false, 0.05f);

```

```

17 LoadAnimation("Sprites/LevelObjects/Player/spr_die@5", "die", true, 0.1f);
18 LoadAnimation("Sprites/LevelObjects/Player/spr_explode@5x5", "explode", false, 0.04f);
19
20 // start with the idle sprite
21 PlayAnimation("idle");
22 SetOriginToBottomCenter();
23 facingLeft = false;
24 }
25
26 public override void HandleInput(InputHelper inputHelper)
27 {
28     // arrow keys: move left or right
29     if (inputHelper.KeyDown(Keys.Left))
30     {
31         facingLeft = true;
32         velocity.X = -walkingSpeed;
33         PlayAnimation("run");
34     }
35     else if (inputHelper.KeyDown(Keys.Right))
36     {
37         facingLeft = false;
38         velocity.X = walkingSpeed;
39         PlayAnimation("run");
40     }
41     // no arrow keys: don't move
42     else
43     {
44         velocity.X = 0;
45         PlayAnimation("idle");
46     }
47
48     // set the origin to the character's feet
49     SetOriginToBottomCenter();
50
51     // make sure the sprite is facing the correct direction
52     sprite.Mirror = facingLeft;
53 }
54
55 void SetOriginToBottomCenter()
56 {
57     Origin = new Vector2(sprite.Width / 2, sprite.Height);
58 }
59
60 }

```

Listing 23.3 shows the code for this first version of Player. Go ahead and type this code into your *Player.cs* file, or copy it from the TickTick2 example project.

We've added a constant for the character's walking speed and a Boolean member variable that says which direction the character is facing. In the constructor of Player, we load all animations that we'll need in the game: animations that show the character standing still ("idle"), running, jumping, celebrating, dying, and exploding. Some of these animations should loop; others should not. (By the way, the idle animation has only one frame and is actually just an ordinary sprite, but it will still work if we treat it as an animation.) We start by showing the idle animation, and we store the fact that the character is initially looking to the right.

We also set the origin to the bottom center of the sprite. This will be useful later on, when we add code to make sure that the character can land nicely on platforms.

The only other method that we need right now is `HandleInput`. Here, we check if the player is pressing the left or right arrow key. If either of those keys is being held down, we play the “run” animation, we set the character’s horizontal speed, and we update the `facingLeft` variable. This code will choose a new animation in each frame, but that’s okay: the `AnimatedGameObject` class won’t restart an animation if it is already being played. Near the end of the `HandleInput` method, we recalculate the character’s origin, because the different sprites might have different heights.

Finally, the current sprite gets mirrored based on the `facingLeft` variable. This variable may not seem very useful at first sight: why don’t we just mirror the sprite whenever the character’s horizontal speed is smaller than zero? Well, one reason is that we need to *remember* the character’s heading for when it goes to the idle state. In that state, the horizontal speed is exactly zero, so it’s unclear whether the sprite should be mirrored if we base everything on the speed alone. With our extra member variable, we can make sure that the idle character will have the same heading as when it was still moving. In later chapters, the `facingLeft` variable will prove to be useful for other things as well.

23.3.2 Adding an Instance to the Game

To see the `Player` class in action, let’s add an instance of it to a level. First of all, it’s useful to give the `Level` class an extra member variable that refers to the `Player` instance:

```
Player player;
```

Next, go to the file `LevelLoading.cs` and look for the `LoadCharacter` method. This method is currently empty, but we can now fill it in to add a `Player` instance at the desired position:

```
void LoadCharacter(int x, int y)
{
    player = new Player();
    player.LocalPosition = GetCellPosition(x, y) + new Vector2(TileWidth / 2, TileHeight);
    AddChild(player);
}
```

We set the player’s position so that it is (horizontally) nicely centered on the starting tile and (vertically) exactly standing on the tile below.



CHECKPOINT: TickTick2

Compile and run the game now. When you load a level, the bomb character will appear at the correct position, and you can control it with the arrow keys.

Note that the character is still only moving horizontally, and not vertically. For example, the character cannot jump yet, and it doesn’t fall down when there is empty space below it. You will add those features (along with other forms of *game physics*) in the next chapter.

23.4 What You Have Learned

In this chapter, you have learned:

- how to create animations by letting objects show different sprites over time;
- how to build an animated game object with multiple animations.

23.5 Exercises

1. *Class Hierarchy of the Engine*

With the `Animation` and `AnimatedGameObject` classes, the structure of our game engine has become a bit more complicated again.

To get a good overview, draw a diagram of the classes and interfaces we've introduced so far. Use lines to connect classes that inherit from each other, just like in the “vehicles” example from Chap. 10.

Chapter 24

Game Physics



So far, our Tick Tick game contains tile-based levels and an animated bomb character that can move left or right depending on keyboard input. This is clearly not enough yet, though: the character should also be allowed to jump onto platforms, fall down when it walks off a platform, stop walking when it runs into a wall, and so on. We should also prevent the character from walking outside the level's boundaries, which is currently still allowed.

In this chapter, you'll add those features by implementing the *physics* of Tick Tick. These physics define how the bomb character interacts with the game world. Because this behavior applies only to the bomb character, you'll implement it mostly inside the `Player` class. However, we'll also discuss general reusable code for calculating if two objects collide. This reusable code is part of the Engine library.

At the end of this chapter, players will have full control over the bomb character, and it will respond correctly to the walls and platforms in a level.

Watch out: implementing game physics correctly can be a very tricky task. The code that we're about to discuss actually took *us* a bit of trial and error as well. Larger game studios often use professional *physics engines* that are very robust, so programmers don't have to re-invent the wheel. For Tick Tick, we don't have this luxury, and we'll have to write a simplified physics system ourselves—which, unfortunately, is one of the most difficult parts of the program to understand.

24.1 Jumping and Falling

First of all, the bomb character should be able to jump and fall. We can implement falling by adding a *gravity* force that pulls the character down. To let the character *jump*, we can simply send it upward at a high speed, and then gravity will make sure that the character follows a nice arc. This is comparable to what we did for the ball of the Painter game, way back in Part II of this book.

Let's start by giving the `Player` class some new constants:

```
const float jumpSpeed = 900; // lift—off speed when the character jumps
const float gravity = 2300; // strength of the gravity force
const float maxFallSpeed = 1200; // maximum vertical speed when the character is falling
```


These constants were chosen by us so that the example levels that we've prepared are interesting to play. If you want to, you can tweak these values to let the character jump higher or fall down more slowly—but let's implement the basics first before worrying about those details.

24.1.1 Applying Gravity

Add a method `ApplyGravity` that updates the character's vertical speed according to gravity:

```
void ApplyGravity(GameTime gameTime)
{
    velocity.Y += gravity * (float)gameTime.ElapsedGameTime.TotalSeconds;
    if (velocity.Y > maxFallSpeed)
        velocity.Y = maxFallSpeed;
}
```

Just like with the ball in the Painter game, the gravity's effect gets scaled by the amount of time that has passed in the last frame. That way, the strength of the gravity does not depend on the framerate. This time, we added a maximum falling speed so that the character is still easy to follow when it's falling a long way down.

Next, give the `Player` class its own `Update` method. For now, it should just call `ApplyGravity` and then call the base version of `Update`, which will (in turn) update the character's position based on its velocity:

```
public override void Update(GameTime gameTime)
{
    ApplyGravity(gameTime);
    base.Update(gameTime);
}
```

We'll add more code to this method later on.

24.1.2 Letting the Character Jump

The character should jump when the player presses the spacebar. But there's a catch: we *only* want to allow a jump when the character is currently standing on the ground. Otherwise, players could bash the spacebar to keep pushing the character up when it's in the air. Let's add another member variable that will keep track of whether or not the character is standing on a surface:

```
bool isGrounded;
```

Set this variable to **true** in the constructor of `Player`. Later in this chapter, you'll write code that gives this variable the correct value in each frame of the game loop, by checking if there is a floor tile below the character. For now, let's assume that the variable always has the correct value.

With this `isGrounded` variable in place, go to the `HandleInput` method and add the following code *after* the **if** and **else** blocks that are already there:

```
if (isGrounded && inputHelper.KeyPressed(Keys.Space))
{
    velocity.Y = -jumpSpeed;
    PlayAnimation("jump");
}
```

This means that when the character is on the ground and the player presses the spacebar, the character will jump up and display the matching animation. Immediately after this code, add code that shows a “falling” animation (which is the second half of the “jump” animation) when the character is not standing on the ground:

```
if (!isGrounded)
    PlayAnimation("jump", false, 8);
```

This will make the character look like it’s falling when it walks off a platform. The second parameter **false** will make sure that the animation only starts if it wasn’t already playing. As a result, the “regular” jump animation will still look smooth.

24.1.3 Finishing Touches and Design Choices

If you think critically about the code so far, you may notice that `HandleInput` always starts by choosing between the “run” and “idle” animations. The “jump” animation gets chosen when the player has just pressed the spacebar, but that will only be true during a single frame. In the frame after that, it will already be overwritten again by the “run” or “idle” animation. To avoid that, we should show the “run” and “idle” animations *only* when the character is standing on the ground. So, add an **if** condition before the lines that play these animations:

```
if (isGrounded)
    PlayAnimation("run");
...
if (isGrounded)
    PlayAnimation("run");
...
if (isGrounded)
    PlayAnimation("idle");
```

If you’ve gotten confused about what to write where, take a look at the `TickTick3` example project for this chapter.

Do you see that the character’s horizontal and vertical movement are completely detached? Jumping and moving sideways are two different things that can be triggered independently from each other. And if the player is pressing multiple keys at the same time, the character can also jump sideways.

By the way, this code even responds to the arrow keys when the character is in the air. As a result, the character can start moving to the left or right without having to stand on the ground. Although it’s impossible in the real world to change your horizontal direction when you’re falling,¹ this does make the game nicer to play.

This wraps up the code for letting the character jump and fall. However, if you compile and run the game now and start a level, the character will immediately start falling down! This is because we have not yet implemented any *collision detection* that lets the character stop falling when it’s standing on a platform.

¹Don’t try this at home!

24.2 General Collision Detection

Detecting collisions between game objects is a very important part of simulating interactive game worlds. Such **collision detection** is used for many different things in games: detecting collisions between the character and walls or floors, detecting if you walk over a power-up, detecting if you collide with a projectile, and so on.

In big games, the program will do a *lot* of collision checking in each frame of the game loop. Therefore, it's a good idea to keep the code for collision detection very simple, so that it doesn't cost too much computation power. A common approach is to simplify the game objects by basic shapes, such as rectangles or circles. In general, these simplified shapes are called **bounding volumes**.² More specifically, you could speak of a bounding *box* or a bounding *circle*. You've actually done this before in the Painter game, perhaps without knowing it. Take a look at this code from the PaintCan class:

```
if (BoundingBox.Intersects(Painter.GameWorld.Ball.BoundingBox))
{
    Color = Painter.GameWorld.Ball.Color;
    Painter.GameWorld.Ball.Reset();
}
```

In this example, you used the BoundingBox property (which returns a rectangle) of the paint can and the ball and checked if these rectangles overlapped. In other words, you've checked for a collision between two bounding boxes!

This section dives deeper into collision detection. This is a rather stand-alone piece of text, so if you're really in a hurry to continue working on Tick Tick, you could also go straight to Sect. 24.3. Of course, we encourage you to study *this* section as well, though.

24.2.1 Collision Detection with Bounding Volumes

In the Engine project, we've written a (small) library for collision detection between bounding shapes. Next to rectangles, it also supports circles.

The CollisionDetection class (in a subfolder with the same name) is a class with only public static methods, so that you can call them from everywhere. Next to checking for intersections between *rectangles*, it can also check for intersections between *circles*, as well as a combination of those two. From now on, whenever we talk about "rectangles," we actually mean *axis-aligned* rectangles: that is, rectangles with perfectly horizontal and vertical sides. (If we allow rectangles to rotate, things become much more difficult, and that's beyond the scope of this book.)

We've given the CollisionDetection class three methods that calculate if there's a collision between a circle and a circle, a rectangle and a rectangle, or a rectangle and a circle. Fig. 24.1 shows these three cases. The corresponding methods of CollisionDetection are all named ShapesIntersect, but they have different types of parameters. In C#, a class is allowed to have multiple methods with the same name, as long as these methods do not have exactly the same types (and number) of parameters. Otherwise, when you call such a method, there's no way for the compiler to know which version you mean.

To make the code of these methods a bit simpler, we've added a Circle struct, which can be used to represent a circle by a center (Vector2 Center) and a radius (**float** Radius). Take a look at that struct in the Engine project if you're interested.

²Technically, the word "volume" is meant for 3D shapes, such as cubes and spheres. In a 2D game world, it's officially better to use the word "area." But let's ignore this for now.

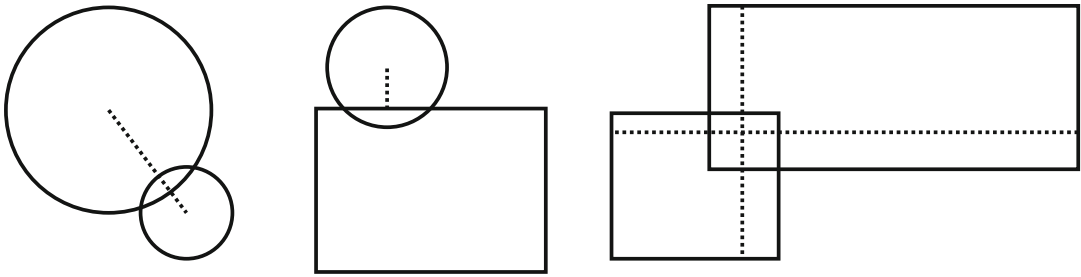


Fig. 24.1 Different types of collisions: circle-circle, circle-rectangle, and rectangle-rectangle

Let's look at the three cases one by one now. To check if a *circle* intersects with another *circle*, we only have to check if the distance between the *centers* of these circles is smaller than (or equal to) the sum of their *radii*. Take your time to understand why this is true, using Fig. 24.1 if necessary. The following method does the job:

```
public static bool ShapesIntersect(Circle circle1, Circle circle2)
{
    return Vector2.Distance(circle1.Center, circle2.Center)
        <= circle1.Radius + circle2.Radius;
}
```

To check if a *rectangle* intersects with another *rectangle*, we have to check if the *x*-ranges *and* the *y*-ranges of the rectangles both overlap. We could explain this in more detail, but the *Rectangle* class of MonoGame actually already has this functionality built in: the *Intersects* method. So, our *CollisionDetection* class can simply call that method:

```
public static bool ShapesIntersect(Rectangle rectangle1, Rectangle rectangle2)
{
    return rectangle1.Intersects(rectangle2);
}
```

To check if a *rectangle* intersects with a *circle*, we have to do some extra work. A good solution is to first calculate the point *P* inside the rectangle that is closest to the circle's center *C*. If *C* already lies inside the rectangle, then *P* is simply equal to *C*. Otherwise, *P* is a point on the *boundary* of the rectangle. We can calculate the coordinates of *P* by *clamping* the coordinates of *C* so that they lie inside the rectangle.

Then, we have an intersection if the distance between *C* and *P* is smaller than (or equal to) the circle's radius. The following method sums it up:

```
public static bool ShapesIntersect(Rectangle rectangle, Circle circle)
{
    Vector2 closestPoint;
    closestPoint.X = MathHelper.Clamp(circle.Center.X, rectangle.Left, rectangle.Right);
    closestPoint.Y = MathHelper.Clamp(circle.Center.Y, rectangle.Top, rectangle.Bottom);

    return Vector2.Distance(circle.Center, closestPoint) <= circle.Radius;
}
```

Take your time to compare this code to the second image of Fig. 24.1.

Finally, the *CollisionDetection* class contains one more method that calculates the overlapping part of two rectangles. This method will be useful for the game physics of Tick Tick later on. To explain how this method works, Fig. 24.2 shows an example of two rectangles that overlap.

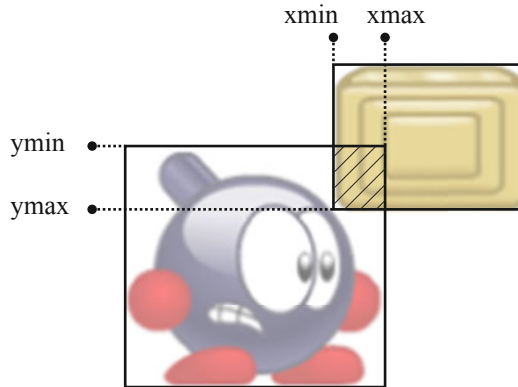


Fig. 24.2 The overlapping part of two rectangles is defined by the minimum and maximum x - and y -coordinates of both rectangles

As you can see in this figure, the *left* boundary of the overlapping part is the largest of the left boundaries of the two rectangles. Likewise, the *right* boundary of the overlapping part is the smallest of the right boundaries of both rectangles. For the top and bottom boundaries, a similar principle holds. This leads to the following method, which constructs a new `Rectangle` object describing the result:

```
public static Rectangle CalculateIntersection(Rectangle rect1, Rectangle rect2)
{
    if (!ShapesIntersect(rect1, rect2))
        return new Rectangle(0,0,0,0);

    int xmin = Math.Max(rect1.Left, rect2.Left);
    int xmax = Math.Min(rect1.Right, rect2.Right);
    int ymin = Math.Max(rect1.Top, rect2.Top);
    int ymax = Math.Min(rect1.Bottom, rect2.Bottom);

    return new Rectangle(xmin, ymin, xmax - xmin, ymax - ymin);
}
```

This method first checks if the two input rectangles really do overlap. If not, it returns a “dummy” rectangle with no width and height. In the other case, it calculates the overlapping part and returns that as a new `Rectangle`.

24.2.2 Limitations of Bounding Volumes

Of course, by simplifying an object to a bounding box or a bounding circle, you *do* lose some detail. For example, representing an object by a circle only gives the “perfect” behavior if the object really is something round, such as a ball. If the object has a more complex shape (such as, say, a paint can), then a simplified bounding volume will never perfectly cover the object. The bounding volume will always be too large or too small.

If a bounding volume is too large, the game might report a collision when the player doesn’t expect it yet. This is especially bad if it causes the player to lose the game. On the other hand, if a bounding volume is too small, the player might see objects overlapping when game doesn’t recognize that yet. This could make it look like the game contains a mistake, which harms the player’s experience as

well. So, whenever you use bounding volumes, it's good to think about their shapes and sizes so that the gameplay doesn't suffer from it.

You could make it possible to give an object *multiple* bounding volumes, but even then, the power of bounding volumes is still limited. It's especially difficult when your objects are *animated*, because then their shape changes over time! At the end of this chapter, we'll look at an alternative solution (*pixel-precise* collision checking), but that's not yet required for letting Tick Tick's bomb character interact with the level.

24.3 Adding Collision Detection to Tick Tick

Now that you understand the basics of collision detection, let's apply it to the Tick Tick game. In each frame of the game loop, the bomb character should check for collisions with wall tiles and platform tiles.

24.3.1 Communication Between the Character and the Level

To make collision detection possible, the Player class somehow needs to have access to the tiles of the level. There are many ways to achieve this, as is usually the case with interaction between game. Just like in Penguin Pairs, we suggest to give the Player class a member variable that refers to the Level in which the character lives:

```
Level level;
```

Update the constructor of the Player class so that it takes a Level as an extra parameter, and make sure that the Level class fills in this parameter (with the value **this**) when creating a **new** Player(...).

The character now has access to the Level object, but we still need to allow Player to ask for the level's tile information. For that, we suggest to give the Level class a method GetTileType that finds and returns the tile type for a certain grid position:

```
public Tile.Type GetTileType(int x, int y)
{
    return tiles[x, y].TileType;
}
```

This way, the Player class can call level.GetTileType(...) to get the type of a particular tile.

The Level class already has a method GetCellPosition that returns the top-left corner of a cell in the game world. Let's also give it a method that does the opposite: for a certain position in the game world, determine in which cell it lies. The following method does exactly that:

```
public Point GetTileCoordinates(Vector2 position)
{
    return new Point((int)Math.Floor(position.X / TileWidth),
                    (int)Math.Floor(position.Y / TileHeight));
}
```

The Player class will use this method soon.

24.3.2 Giving the Character a Custom Bounding Box

For detecting whether our character collides with a tile, we'll give the character a rectangular *bounding box*. We'll say that the character *collides* with a tile if this bounding box overlaps with the rectangle containing that tile.

The `SpriteGameObject` class already has a `BoundingBox` property that returns the bounding box of the full sprite. However, for the bomb character, this bounding box is too big for collision detection: there's quite some whitespace around the character in each sprite. If we use this bounding box directly, then the character will (for example) keep floating in the air when its feet have already moved off a platform. As an alternative, let's give the character a property `BoundingBoxForCollisions` that returns an adjusted version of `BoundingBox`:

```
Rectangle BoundingBoxForCollisions
{
    get
    {
        Rectangle bbox = BoundingBox;
        bbox.X += 20;
        bbox.Width -= 40;
        bbox.Height += 1;
        return bbox;
    }
}
```

This property ignores 20 pixels on the left and right sides of the sprite, to account for the whitespace. Also, it adds one extra pixel *below* the sprite. This will be very useful: when the character is standing exactly on a platform, then its sprite only barely touches the platform (without really intersecting it). In that case, we want to prevent the character from falling. By making the character's bounding box one pixel larger at the bottom, we allow the game to easily detect those kinds of collisions.

24.3.3 Adding the *HandleTileCollisions* Method

Our character should check for collisions in each frame of the game loop. We'll add a new method `HandleTileCollisions` for that, and we'll call that method immediately *after* the character's position has been updated.

When checking for collisions, it's useful to remember the character's *previous* position as well. This makes it easier to check where the character was coming from, which is sometimes useful for handling a collision in the correct way. So, first change the `Update` method so that it looks like this:

```
public override void Update(GameTime gameTime)
{
    Vector2 previousPosition = localPosition;

    ApplyGravity(gameTime);
    base.Update(gameTime);

    HandleTileCollisions(previousPosition);
}
```

In other words, before applying gravity and moving the character, we remember the character's current position. After the call to **base.Update**, this can be seen as the character's *previous* position. This previous position is a parameter for the `HandleTileCollisions` method that we have to create next:

```
void HandleTileCollisions(Vector2 previousPosition)
{
    // TODO: handle collisions
}
```

This method should check for collisions between the character and the level's tiles. The result may affect the value of the `isGrounded` variable. Inside this method, start by setting the variable to **false**:

```
isGrounded = false;
```

Later in the method, we'll set it back to **true** again if we discover that the character is currently standing on a tile.

Watch out: there's a lot of code coming up! This is the tricky part of the program that we warned you about earlier.

24.3.4 Finding the Range of Tiles to Check

To check for all possible collisions between the character and the tiles, we *could* loop over all tiles in the level and check each individual tile. However, many of these tiles will certainly be too far away from the player. Luckily, we can do something smarter, by making use of the fact that all tiles are nicely aligned in a grid. The idea here is to simply skip all the tiles that are too far away.

Instead of checking all the tiles, let's only look at the tiles that overlap with the character's current bounding box. We'll add a margin of 1 tile on each side, just to be sure that we don't miss anything due to rounding errors. The following code retrieves the top-left and bottom-right tile to check:

```
Rectangle bbox = BoundingBoxForCollisions;
Point topLeftTile = level.GetTileCoordinates(new Vector2(bbox.Left, bbox.Top))
    - new Point(1,1);
Point bottomRightTile = level.GetTileCoordinates(new Vector2(bbox.Right, bbox.Bottom))
    + new Point(1, 1);
```

Place this code inside the `HandleTileCollisions` method. Here, we're copying the bounding box into another variable `bbox` to ensure that we only calculate the bounding box once per frame. If we would type `BoundingBoxForCollisions` everywhere, the program would recalculate the bounding box each time it sees that word.

Now, instead of looping over *all* tiles, we can just loop over the tiles in the chosen range:

```
for (int y = topLeftTile.Y; y <= bottomRightTile.Y; y++)
{
    for (int x = topLeftTile.X; x <= bottomRightTile.X; x++)
    {
        // check for collisions with one tile
    }
}
```

What remains is to fill in the code that detects collisions between the character and one individual tile.

Acceleration Structures — Just now, we’ve made collision detection with tiles more efficient by ignoring the tiles that are definitely too far away from the player. We’ve used the grid of tiles as an *acceleration structure*: a special way of representing the game world so that certain questions can be answered more quickly. In this case, the grid-based layout of the game world makes collision detection much easier.

Acceleration structures are very important in general. They’re used really often in computer graphics and other areas of computer science. For example, big 3D game engines (such as Unity and the Unreal Engine) can render complicated game worlds in real time, because they can cleverly ignore the objects that the camera doesn’t see. Without a good acceleration structure, it would be too difficult to find out which objects can be ignored. So, acceleration structures are often crucial for letting a big game run smoothly.

24.3.5 Dealing with Level Boundaries

The range of tiles calculated by `HandleTileCollisions` might include grid coordinates that lie *outside the level*. For instance, if the character is standing at tile (0, 10), this range will include the grid position (−1, 10) to the left of the character, but that grid position is not part of the level itself. Currently, the `GetTileType` method of `Level` cannot handle this.

Checking grid positions that lie outside the level may sound like a mistake, but it can actually be useful. If we handle these “invalid” positions in the right way, we can prevent the character from walking off the side of the level. That’s a nice bonus effect!

To improve the `GetTileType` method, let’s simply *pretend* that there are tiles outside the level’s boundaries. If the given *x*-coordinate or *y*-coordinate is too high or too low, we’ll just return one of the tile types, even though there’s not really a tile at that position.

For the horizontal direction, let’s pretend that there are *wall tiles* on the far left and right side of a level. This will prevent the character from walking outside of the level’s boundaries (and then falling into an endless invisible pit³). At the beginning of the `GetTileType` method, simply return the `Wall` type if the *x*-coordinate is too low or too high to fit inside the level:

```
if (x < 0 || x >= tiles.GetLength(0))
    return Tile.Type.Wall;
```

For the vertical direction, let’s pretend that there are *empty tiles* above or below the level. For those *y*-coordinates, return the `Empty` type:

```
if (y < 0 || y >= tiles.GetLength(1))
    return Tile.Type.Empty;
```

This choice will allow the character to still perform a “full” jump even if it’s near the top of the level. In other words, our levels will have invisible side walls, but *not* an invisible ceiling. It also allows the character to fall into a pit at the *bottom* of the level. (Later, we’ll have to add code to detect that: if the character has disappeared into such a pit, then it should die.)

³Another thing that you probably shouldn’t try at home.

If both of these **if** checks fail, then the grid coordinates lie inside the level. In that case, you can safely ask for the type of the corresponding tile, as before:

```
return tiles[x, y].TileType;
```

This final version of `GetTileType` will make sure that the character is constrained inside the level in a logical way.

24.3.6 Collision Detection with a Single Tile

Now, let's go back to the `HandleTileCollisions` method of the `Player` class. Inside the nested **for** loop, we should check for collisions with a single tile. Start by obtaining the type of this tile. If this tile has the type `Empty`, then there is no obstruction there, and we don't have to do anything:

```
Tile.Type tileType = level.GetTileType(x, y);
if (tileType == Tile.Type.Empty)
    continue;
```

It's been a while since we've used the keyword **continue**. Remember from Chap. 9 that this instruction lets the program jump to the next loop iteration, skipping the rest of the code in the *current* iteration. You can usually do without this keyword, but it can be a nice trick in a complicated loop such as the one we have here. Just like how a *method* can use **return** instructions to let the method stop earlier in special cases, a *loop* can use **continue** instructions to skip iterations that aren't interesting.

Next up, remember that platform tiles are special tiles that the player can jump onto from below. So, if this tile has the type `Platform`, then we only have to do something if the character is standing on *top* of the platform. Otherwise, we act as if the tile is empty.

The following code does this, by checking if the character is standing on top of the tile *and* was standing on top of the tile in the previous frame. Just checking the character's *current* position is not enough: if we're jumping through the tile from below and we've now reached above it for the first time, we don't want to ignore the tile.

```
Vector2 tilePosition = level.GetCellPosition(x, y);
if (tileType == Tile.Type.Platform
    && localPosition.Y > tilePosition.Y && previousPosition.Y > tilePosition.Y)
    continue;
```

There are other ways to achieve the same effect, but this code does the job.

If the program passes the tests so far, we're apparently dealing with a tile that should be treated as an obstacle. We should now check if there is really a collision between the character and the tile. If not, we (again) don't have to do anything. The following code does exactly that:

```
Rectangle tileBounds = new Rectangle(
    (int)tilePosition.X, (int)tilePosition.Y, Level.TileWidth, Level.TileHeight);
if (!tileBounds.Intersects(bbox))
    continue;
```

We're creating our own `Rectangle` here to represent the tile's bounding box. This is necessary because the tile might not really exist: if the tile lies outside the level, then we're only *pretending* that it exists, so then there's not really a `Tile` object with a `BoundingBox` property that we can use.

24.3.7 Responding to a Collision with a Tile

If the program passes all the checks we've written so far, then there's really a collision between the character and the tile that we're currently looking at. But how should we *respond* to such a collision? The tile could be a floor, a ceiling, or a wall to the left or right. How do we decide which of these four cases we're dealing with?

We'll start by asking for the overlapping part of the two bounding boxes:

```
Rectangle overlap = CollisionDetection.CalculateIntersection(bbox, tileBounds);
```

This rectangle has a width and a height, which indicate the amount of overlap in the horizontal and vertical direction. Both numbers are larger than zero; otherwise, there wouldn't be any overlap. So, the bounding boxes overlap in the *x*-direction *and* in the *y*-direction.

Now, let's assume that the *smallest* of these two values is the interesting one. For example, if the character is falling onto the floor, then the *y*-overlap will be very small, but the *x*-overlap could be quite big, because the character might cover the full width of the tile. By contrast, if we're walking into a wall, the *x*-overlap will be smaller than the *y*-overlap. This leads to the following case distinction:

```
if (overlap.Width < overlap.Height)
{
    // handle a horizontal collision
}
else
{
    // handle a vertical collision
}
```

If the collision is *horizontal*, we should basically prevent the character from moving in its desired direction. To do that, we can revert the *x* component of the character's position to its previous value—as if the character hasn't moved horizontally at all in this frame. We also want to set the character's horizontal speed to zero:

```
localPosition.X = previousPosition.X;
velocity.X = 0;
```

However, we only want to do these things in certain cases, namely:

1. if the character is moving to the *right*, but it's standing to the *left* of the tile;
2. if the character is moving to the *left*, but it's standing to the *right* of the tile.

To check if the character is to the left or to the right of a tile, the center of the character's bounding box is a nice reference point. So, the code for handling horizontal collisions looks like this:

```
if ((velocity.X >= 0 && bbox.Center.X < tileBounds.Left) || // right wall
    (velocity.X <= 0 && bbox.Center.X > tileBounds.Right)) // left wall
{
    localPosition.X = previousPosition.X;
    velocity.X = 0;
}
```

If the collision is *vertical*, the idea is similar, but we should (of course) do vertical checks instead of horizontal ones. Also, we might have to do some extra work to let the character snap nicely onto the floor.

If the character is moving down and its center is above the tile, then we've fallen onto the floor. We should set the `isGrounded` variable to **true**, set the vertical speed to zero, and make sure that the character gets nicely aligned with the top of the floor tile. The following code does this:

```
if (velocity.Y >= 0 && bbox.Center.Y < tileBounds.Top)
{
    isGrounded = true;
    velocity.Y = 0;
    localPosition.Y = tileBounds.Top;
}
```

Note: we're "manually" setting the *y*-coordinate of the character so that it matches the top of the tile. If we would simply revert to the *previous y*-coordinate (just like in the horizontal case), the character would stop in midair just above the tile, depending on how fast it was falling. Snapping the character onto the tile is a better choice in this case.

If the character is moving up and its center is below the tile, then we've bumped our head into a ceiling. In that case, reverting to the previous *y*-coordinate *is* good enough, because we don't have to snap precisely onto the ceiling. We should also stop moving vertically, so that we can immediately fall down in the next frame. Here's what the code looks like:

```
else if (velocity.Y <= 0 && bbox.Center.Y > tileBounds.Bottom)
{
    localPosition.Y = previousPosition.Y;
    velocity.Y = 0;
}
```

This concludes all cases and therefore the `HandleTileCollisions` method as a whole. The game is now playable in the sense that you can walk and jump through a level. Feel free to try it out!

24.3.8 Dealing with False Positives

If you test the game now by walking and jumping through some levels, you may notice that the game physics are not entirely perfect yet. In particular, a horizontal bump into a high wall (such as the right edge of the level) is sometimes accidentally treated as a vertical collision. In those cases, the character is horizontally bumping into multiple tiles at the same time, with a small *x*-overlap for each of them. But for one of those tiles, the *y*-overlap is even smaller, so *that* collision is treated as *vertical* (a floor or a ceiling).

Apparently, choosing the smallest of the two values is not the perfect solution for deciding if a collision is horizontal or vertical. Sometimes, a collision is so small that we're better off ignoring it completely. By trying out a number of possible fixes, we noticed that the problem can be resolved with these two steps:

- In the "floor" case, add `&& overlap.Width > 6` to the **if** condition. This will ignore "floor" detections when we're very close to the edge of a tile.
- In the "ceiling" case, add `&& overlap.Height > 2` to the **if** condition. This will ignore "ceiling" detections when we're only barely bumping our head into a tile.

This solution might not sound very logical: why do we use exactly these numbers, and why do we use `Width` for one part and `Height` for the other? Well, we must admit that this solution is a bit "magical" and that we had to find it via trial and error. The hard truth is that game programming (especially

around physics and geometry) sometimes involves tweaking and patching, no matter how good the rest of your code is. Feel free to play around with these checks to see how they affect the game.

Compile and run the game again. The character should now respond correctly to collisions with tiles.

24.4 Adding Slippery Ice and Friction

There's one more missing gameplay element of Tick Tack that's strongly related to game physics: dealing with icy and hot surfaces in a level. In particular, when the bomb character is standing on an icy tile, they should move in a more "slippery" way. To implement this nicely, we have to change our game physics a little bit.

24.4.1 Detecting the Type of Surface Below the Player

The first step is to *detect* the type(s) of surface on which the player is standing. (Note that the bomb character could be standing on multiple tiles at the same time, so also on multiple *surface types*.) To implement this detection, start by giving the Player class two more member variables:

```
bool standingOnIceTile, standingOnHotTile;
```

and set them both to **false** in the constructor of Player. In each frame of the game loop, we'll check if the player is *currently* standing on an ice tile or a hot tile, and we'll store the result in these two variables. The `HandleTileCollisions` method is a good place for performing these checks, because that method already checks if the character is standing on a tile in general.

Just like with the `isGrounded` variable, we'll always start by assuming that the character is *not* standing on a special tile, until we discover otherwise. So, at the beginning of the `HandleTileCollisions` method, set both member variables to **false**:

```
standingOnIceTile = false;  
standingOnHotTile = false;
```

Later in the method, deep inside the nested **for** loop, look for the part of the code that detects a floor tile. Inside the same **if** block that already sets `isGrounded` to **true**, we should now also check for the surface type of the tile. If it's a hot tile or an ice tile, the corresponding `standingOn...` variable should be set to **true**:

```
Tile.SurfaceType surface = level.GetSurfaceType(x, y);  
if (surface == Tile.SurfaceType.Hot)  
    standingOnHotTile = true;  
else if (surface == Tile.SurfaceType.Ice)  
    standingOnIceTile = true;
```

To make this code work, you'll have to give the Level class a `GetSurfaceType` method. Given a pair of tile coordinates, this method should return the surface type of the corresponding tile or `Normal` if the coordinates lie outside the level. Other than that, the method is quite similar to `GetTileType`, so we'll leave it to you to fill in the details. You can find the answer in the `TickTack3` example project, of course.

24.4.2 Adding Friction to the Game Physics

Now, the idea is that walking on ice should prevent the character from immediately doing what the player wants. If the player stops pressing an arrow key or starts pressing the key for the opposite direction, the character should keep sliding for a bit.

To achieve this effect, we should first change the `HandleInput` method so that pressing/releasing the arrow keys doesn't change the character's *real* velocity, but only its *desired* velocity. What we *do* with this desired velocity will then depend on the surface on which the character is walking.

Start by giving the `Player` class another member variable:

```
float desiredHorizontalSpeed;
```

Then change the `HandleInput` method so that it sets `desiredHorizontalSpeed` instead of `velocity.X`. There are three lines of code that you should change here. (For the vertical speed, you don't have to make any changes.)

At the beginning of the `Update` method, we now want to update `velocity.X` so that it moves towards `desiredHorizontalSpeed`, but the surface type should determine how quickly this happens. On an ice tile, we only want to change `velocity.X` a little bit, but on a normal tile, we want to change it more strongly. The recipe for this is to update the velocity in the following way:

```
velocity.X += (desiredHorizontalSpeed - velocity.X) * multiplier;
```

where `multiplier` is a number between 0 and 1 that we haven't calculated yet. If this number is 1, the velocity will be instantly set to `desiredHorizontalSpeed`. If it's 0, the desired speed will be ignored entirely. If it's something in-between, the character will try to use the desired speed, but it won't be able to apply it *completely*, for example, because the floor is slippery.

But how should we calculate a meaningful value for `multiplier`? We can do that by adding a simple *friction* system to the game. Our system won't exactly match how friction works in real-world physics, but it will be good enough for this game.⁴

To implement friction, start by adding three more constants to the `Player` class:

```
const float iceFriction = 1;
const float normalFriction = 20;
const float airFriction = 5;
```

A low value should lead to a low multiplier. Note that we're defining a low friction value for ice and a higher friction value for normal surfaces. So, these numbers will indeed have the effect that ice is more slippery than a normal surface. (We've added air friction as a bonus, to prevent players from turning around instantly in midair. It's still possible to turn around, but it will happen in a more subtle way.)

In the `Update` method, we should first determine which friction value to use in the current frame, based on the character's current situation:

```
float friction;
if (standingOnIceTile)
    friction = iceFriction;
else if (isGrounded)
    friction = normalFriction;
else
    friction = airFriction;
```

⁴The real world doesn't contain walking bombs with faces either, so there's room for creativity.

To convert the friction to a meaningful value for multiplier, we should take into account how much time has passed in this frame. That way, the code will be framerate-independent. Furthermore, we want to make sure that multiplier always lies between 0 and 1, because otherwise we'll get strange results. In summary, the following line of code converts friction to a useful multiplier:

```
float multiplier = MathHelper.Clamp(
    friction * (float)gameTime.ElapsedGameTime.TotalSeconds, 0, 1);
```

Now that you've calculated the multiplier, you can use it to update the velocity, as we've indicated before:

```
velocity.X += (desiredHorizontalSpeed - velocity.X) * multiplier;
```

And finally, when velocity.X has a very small value, it's useful to manually set it to zero:

```
if (Math.Abs(velocity.X) < 1)
    velocity.X = 0;
```

Otherwise, the character will always keep sliding a little bit, even when the player doesn't see it. Stopping the character will come in handy in the next chapter, where we'll add enemies that check if the player is currently moving.



CHECKPOINT: TickTick3

If you recompile and run the game now, the character will move differently when it's on ice or in the air.

Congratulations: you've now added all physics-related code to the game! Of course, there are more gameplay elements that we still need to add, such as the *enemies* and the ability to finish a level. You'll deal with those elements in the next two chapters.

24.5 Pixel-Precise Collision Detection

We've explained in this chapter that bounding volumes are very useful but that they have limitations in terms of precision. A bounding volume is always an *approximation* of the actual game object, unless that game object is a perfect circle or rectangle. In 2D games, it's sometimes useful to check if there's an intersection between two sprites in a *pixel-precise* way. This means that we check if there is an overlap between any of the sprites' nontransparent pixels. This gives a more honest answer to the question "do two objects collide?", which can be important for some aspects of your game.

Pixel-precise collision detection is a much more expensive operation⁵ than simply checking if two rectangles or circles overlap. If you use it way too often, your game might slow down. However, it's still interesting to understand how this works.

In this section, you don't have to write any code yourself. We will just explain some parts of the Engine project that we've already prepared for you. In the next chapter, you'll use this pixel-precise collision to check if the character touches an *enemy*. For most other purposes, we'll stick to bounding volumes, because we don't need more precision.

⁵For programmers, "expensive" means "costing a lot of computation power." For managers, "expensive" means "costing a lot of *money*." This can lead to fun misunderstandings on the workflow.

24.5.1 Getting and Storing Transparency Information

To prepare for collision detection, we should store which pixels of a sprite are transparent. You might remember from the Jewel Jam game (where we added glitters to sprites at nontransparent pixels) that the `Texture2D` class has a `GetData` method that gets the “raw” pixel values of a sprite. If you want to know if an individual pixel is transparent, you *could* find that out via `GetData`—but that method is really meant for getting large chunks of pixels at the same time. It will cost quite a lot of time if you call this method over and over again for each pixel. Instead, it’s better to let a sprite *precompute* its transparency information, by calling `GetData` only once and storing the result somewhere.

In the `SpriteSheet` class, you may have already noticed this member variable:

```
bool[,] pixelTransparency;
```

This array stores which pixels of the sprite sheet are transparent. For each pixel coordinate, it stores **true** if the corresponding pixel is completely transparent and **false** if it is not. In the constructor of `SpriteSheet`, after we’ve loaded the sprite, we call `GetData` (once) to get all pixel data:

```
Color[] colorData = new Color[sprite.Width * sprite.Height];
sprite.GetData(colorData);
```

Based on this data, we then fill the `pixelTransparency` array with values that depend on whether or not a pixel’s *alpha* component is zero:

```
pixelTransparency = new bool[sprite.Width, sprite.Height];
for (int i = 0; i < colorData.Length; ++i)
    pixelTransparency[i % sprite.Width, i / sprite.Width] = colorData[i].A == 0;
```

So, we convert detailed pixel information (`colorData`) to a simpler version (`pixelTransparency`) that only stores whether pixels are transparent. By the way, the `GetData` method only works with 1D arrays, but our `pixelTransparency` array is 2D, because that’s easier to work with. Therefore, the code above does some extra work to split a single array index *i* into separate *x* and *y* components.

Next up, `SpriteSheet` has a method `IsPixelTransparent` that returns whether or not a given pixel of the sprite is transparent:

```
public bool IsPixelTransparent(int x, int y)
{
    int column = sheetIndex % sheetColumns;
    int row = sheetIndex / sheetColumns % sheetRows;
    return pixelTransparency[column * Width + x, row * Height + y];
}
```

It simply looks up the answer in the `pixelTransparency` array, after some coordinate conversions. Because a sprite *sheet* can contain multiple sprites in one image, we need to “offset” the given pixel coordinates by the part of the sprite sheet that we’re currently using.

24.5.2 Checking for Overlap Between SpriteGameObject Instances

To check for collisions between sprites, the `SpriteGameObject` class contains a method with the following header:

```
public bool HasPixelPreciseCollision(SpriteGameObject other)
```


This method checks whether or not a `SpriteGameObject` instance has overlapping nontransparent pixels with another `SpriteGameObject`. It starts out by calculating how the bounding boxes of the two objects overlap:

```
Rectangle b = CollisionDetection.CalculateIntersection(BoundingBox, other.BoundingBox);
```

Remember from Sect. 24.2 that the `CalculateIntersection` method returns an empty rectangle if there's no intersection at all. So, after this instruction, we can use a nested **for** loop to iterate over all pixels in that overlapping part:

```
for (int x = 0; x < b.Width; x++)
{
    for (int y = 0; y < b.Height; y++)
    {
        // check pixel (x,y)
    }
}
```

and if the rectangles didn't overlap at all, this loop will simply do nothing.

To check a single pixel in both objects, we have to translate the values of `x` and `y` so that they become the correct pixel coordinates of the two sprites. For this, we have to take the position and origin of both objects into account:

```
int thisX = b.X - (int)(GlobalPosition.X - Origin.X) + x;
int thisY = b.Y - (int)(GlobalPosition.Y - Origin.Y) + y;
int otherX = b.X - (int)(other.GlobalPosition.X - other.origin.X) + x;
int otherY = b.Y - (int)(other.GlobalPosition.Y - other.origin.Y) + y;
```

We should then check pixel (`thisX`, `thisY`) of this object and pixel (`otherX`, `otherY`) of the other object. If both of these pixels are nontransparent, then we report a collision:

```
if (!sprite.IsPixelTransparent(thisX, thisY)
    && !other.sprite.IsPixelTransparent(otherX, otherY))
    return true;
```

Finally, if we reach the very end of the method without ever finding a collision, then there is apparently no collision between any pixels of the two sprites. We then return the value **false**.

The `SpriteGameObject` class also contains a variant of `HasPixelPreciseCollision` that takes a *rectangle* as a parameter. This is a simplified version of the same method: it only takes the transparency of the *current* object into account, and it assumes that the rectangle represents a fully solid object. Apart from that, it works exactly the same.

This concludes the chapter about game physics and collision detection. Next up, let's add enemies to the game!

24.6 What You Have Learned

In this chapter, you have learned:

- how to let a game character jump and fall;
- how to calculate collisions in games and respond to them;
- how a tile-based level structure can lead to faster collision checking;
- that implementing robust game physics is a complicated task, sometimes requiring trial and error;
- how to simulate friction and different surface types;
- how to perform pixel-precise collision detection.

24.7 Exercises

1. *Bounding Volumes*

What is a bounding volume? What are the main advantages and disadvantages of using bounding volumes for game physics?

2. *Collision Detection Between Rectangles*

Implement your own method for collision detection between two (axis-aligned) rectangles. That is, write your own code instead of calling the `Intersects` method of `Rectangle`. *Hint*: two rectangles overlap when they are overlapping in the x -direction *and* in the y -direction.

Chapter 25

Intelligent Enemies



You may remember from Chap. 22 that the Tick Tick game should contain several types of enemies: rockets that fly through the screen, flames that walk over platforms, a spark that drops down and electrifies the player, and a turtle onto which the player can jump. In this chapter, you'll add these enemies to the game, and you'll implement their behavior.

In a way, these enemies show some sort of **artificial intelligence** or "AI": they should do things depending on the game world's current status, such as the player's position in the level, the time that has passed, and so on. Usually, the behavior of enemies in a game should not be *too* intelligent: the player should still be able to beat the game without getting too frustrated.

Artificial intelligence is a huge topic that's getting more and more important in computer games as well. Many books have been written about it, and new techniques are being developed as we speak. Diving deeply into AI would be way too advanced for this book. Luckily, the behavior of the enemies in Tick Tick is quite simple, and you'll be fine with the programming concepts you already know. In particular, you will use *inheritance* to create different variants of an enemy with their own specific behavior. Because none of this is really new material, we'll go through the details pretty quickly.

At the end of this chapter, you'll have programmed the behavior of all enemies in the game, and the player will be able to interact with them.

25.1 A Simple Moving Enemy

One of the most basic enemies in Tick Tick is the *rocket*. A rocket flies from one side of the screen to the other side and then reappears after it has left the screen. If the bomb character touches the rocket, then the character instantly dies. Let's implement this enemy first and see how we should change the rest of the program to make this possible.

25.1.1 Preparing the Level and Player Classes

In a level file, we indicate the starting position of a rocket enemy with the letter "R." For example, here's a level that contains many rockets on both sides:

```
Many, many, many, many, many rockets...  
.....  
R..W.....X....
```

```

. . . - - . W . . . . . - - . . .
. . . . W . - - . . . . . W . . R
. . . - - . . . . . . . . - - . . .
R . . W . . . . . W . . . W . . .
. . . - - . . . . . . . . - - . . .
. . . . W . . . . . . . . W . . .
. . . - - . . . . . W . . - - . . .
R . . W . . . . . - - . W . . .
. . . - - . . . . . . . . - - . . .
. . . . W . . . . . . . . W . . R
. . . - - . . . . . . . . - - . . .
. 1 . . . . . . . . . . . . . . .
##### . ##### . #####

```

In *LevelLoading.cs*, add the following method:

```

void LoadRocketEnemy(int x, int y)
{
    Rocket r = new Rocket(this, GetCellPosition(x, y), x != 0);
    AddChild(r);
}

```

and update the `AddTile` method so that it calls `LoadRocketEnemy` when it receives the symbol “R”. As you can see, this new method creates an instance of a class `Rocket` that we have yet to create. The constructor of `Rocket` will have three parameters: a reference to the level (a trick that we also used for the `Player` class), a starting position, and a Boolean value that indicates whether or not the rocket should move to the left.

This last parameter depends on the position of the rocket in the level. We’ll assume that a rocket enemy always starts on the left side *or* the right side of a level. If it starts on the left side, it has to move to the right, and vice versa.

The rocket enemy (and all other enemies too, actually) will often want to check where the bomb character is located in the game world. After all, they want to check for collisions with the player, or they may even adapt their behavior depending on the player’s position. To prepare for that, go to the `Level` class and replace the player member variable by a public property `Player` with a private **set** component. Change the rest of the class where needed, by following the error messages that Visual Studio shows.

Touching an enemy will cause the bomb character to die. We won’t implement this “dying” behavior just yet, but it’s good to already give the `Player` class a method with the following header:

```

public void Die()

```

You can leave this method empty for now.

As a final preparation step, it’s useful if our upcoming `Rocket` class can ask how large the level is. After all, the rocket should reset itself when it has moved outside the level’s boundaries. Therefore, give the `Level` class a read-only property `BoundingBox` that returns the following expression:

```

new Rectangle(0, 0, tiles.GetLength(0) * TileWidth, tiles.GetLength(1) * TileHeight)

```

25.1.2 Overview of the *Rocket* Class

Let’s work on the `Rocket` class now. In Visual Studio, start by adding a folder “Enemies” inside the “LevelObjects” folder. This folder will contain the classes for all enemies in the game. Next, add a

new Rocket class to this folder. Our version of this class is shown in Listing 25.1. You can find this code in the TickTick4 project as well. We'll go over the details here, but as usual, you're free to make changes if you want to give it a try.

Listing 25.1 The Rocket class

```

1 using Engine;
2 using Microsoft.Xna.Framework;
3
4 /// <summary>
5 /// Represents a rocket enemy that flies horizontally through the screen.
6 /// </summary>
7 class Rocket : AnimatedGameObject
8 {
9     Level level;
10    Vector2 startPosition;
11    const float speed = 500;
12
13    public Rocket(Level level, Vector2 startPosition, bool facingLeft)
14        : base(TickTick.Depth_LevelObjects)
15    {
16        this.level = level;
17
18        LoadAnimation("Sprites/LevelObjects/Rocket/spr_rocket@3", "rocket", true, 0.1f);
19        PlayAnimation("rocket");
20        SetOriginToCenter();
21
22        sprite.Mirror = facingLeft;
23        if (sprite.Mirror)
24        {
25            velocity.X = -speed;
26            this.startPosition = startPosition + new Vector2(2*speed, 0);
27        }
28        else
29        {
30            velocity.X = speed;
31            this.startPosition = startPosition - new Vector2(2 * speed, 0);
32        }
33        Reset();
34    }
35
36    public override void Reset()
37    {
38        // go back to the starting position
39        LocalPosition = startPosition;
40    }
41
42    public override void Update(GameTime gameTime)
43    {
44        base.Update(gameTime);
45
46        // if the rocket has left the screen, reset it
47        if (sprite.Mirror && BoundingBox.Right < level.BoundingBox.Left)
48            Reset();
49        else if (!sprite.Mirror && BoundingBox.Left > level.BoundingBox.Right)
50            Reset();
51
52        // if the rocket touches the player, the player dies
53        if (HasPixelPreciseCollision(level.Player))

```

```

54     level.Player.Die();
55 }
56 }

```

- As a rocket has an animated sprite, the class inherits from `AnimatedGameObject`.
- In the constructor, the rocket stores a reference to the level. It then loads and plays its only animation, which you can find in the example assets. Because this sprite shows a right-facing rocket, the sprite needs to get mirrored if the rocket is going to move to the left. Next, the starting position gets updated a little bit, so that the rocket starts outside the screen and comes into view a bit later. Then, the velocity gets set so that the rocket moves horizontally at a speed of 500 pixels per second. Finally, the `Reset` method gets called, which will move the rocket to its starting position.
- In the `Reset` method, the rocket needs to go back to its starting position. It will continue moving in the same direction, so that it looks like it “wraps around” the level.
- As usual, the `Update` method defines the behavior that a rocket should do in each frame of the game loop. In this case, the rocket first calls `base.Update` to apply its velocity. Next, it should check if it has moved out of the screen; we can use the new `BoundingBox` property of the `Level` class for this. Remember that there are two different cases, depending on whether the rocket is moving to the left or to the right. Finally, we should check if the rocket collides with the player and (if so) call the `Die` method of the player object.

If you compile and run the game now, you can see the rockets in action in levels 3, 4, 6, 7, 9, and 10. However, remember that the `Die` method of the `Player` class is still empty, so the bomb character cannot *really* interact with enemies yet.

25.2 Enemies with Timers

Next up, let’s look at two slightly more complicated enemies: the turtle and Sparky. These enemies both have two *phases*: an “idle” phase in which the player can safely touch them and a “dangerous” phase in which touching the enemy causes the player to die. The classes of these enemies use timers to switch between phases.

For the remaining enemy classes, we will no longer copy all source code into this book. You can find our version of the code in the “Enemies” folder of the `TickTick4` project. We’ll give you a short overview of each class, and you’ll have the choice between two options: copying our code directly into your project, or trying to write it yourself based on our hints. Good luck!

25.2.1 The *Turtle* Class

The `Turtle` class describes the turtle enemy. Our version of the code should be well understandable at this point in the book. Here’s an overview of what the class does:

- The turtle can show two animations: “idle” and “sneeze.” It switches between these two animations every 5 s, via a timer variable (`float timer`). The member variable `bool sneezing` keeps track of which animation is currently playing.
- The `Reset` method starts playing the “idle” animation, initializes the timer, and sets the object’s origin so that the sprite will be nicely drawn on a platform.

- The Update method updates the timer with the number of seconds that has passed. If enough time has passed now, it switches the value of the variable sneezing, which makes sure that a different animation will start playing in the next frame.
- The Update method also checks for collisions with the player, using the same Player property of Level that we also used in the Rocket class. If the player touches this turtle while the turtle's spikes are out, the player dies. If the player falls down onto this turtle while its spikes are *not* out, the player will get launched up into the air with a certain speed.
- The HasSpikesOut property returns **true** if the turtle is playing the “sneeze” animation *and* this animation is at a frame that actually shows spikes. If you look at the sprite *spr_sneeze@9*, you'll see that the spikes are visible from the third frame onward.

To make this code work, you'll have to add two things to the Player class. First of all, add a property `IsFalling` that returns whether or not the player is currently falling down:

```
public bool IsFalling
{
    get { return velocity.Y > 0 && !isGrounded; }
}
```

Second, add a method `Jump` that lets the character jump up with a certain speed:

```
public void Jump(float speed = jumpSpeed)
{
    velocity.Y = -speed;
    // play the jump animation; always make sure that the animation restarts
    PlayAnimation("jump", true);
}
```

Now that you have this separate method for jumping, it's nicer if you also call this new method when the player presses the spacebar. Go ahead and make that change yourself.

To add the turtle to the game world, add the following method to *LevelLoading.cs*:

```
void LoadTurtleEnemy(int x, int y)
{
    Turtle t = new Turtle(this);
    t.LocalPosition = GetCellBottomCenter(x,y);
    AddChild(t);
}
```

and make sure to call it when reading the symbol “T”. `GetCellBottomCenter` is a helper method that you should add as well, which calculates and returns the bottom-center position of a given tile:

```
Vector2 GetCellBottomCenter(int x, int y)
{
    return GetCellPosition(x, y + 1) + new Vector2(TileWidth / 2, 0);
}
```

Using this method in `LoadTurtleEnemy` will place the turtle at the bottom center of its tile, so that it rests nicely on the tile below. You can use the same `GetCellBottomCenter` method when loading the *player*, by the way.

Compile and run the program again. You should now be able to see the turtle in action in levels 7 and 9.

25.2.2 The *Sparky* Class

Next up is “Sparky,” another relatively simple enemy that uses timers to switch between states. This enemy starts by floating in the air. After some time, it drops down and becomes electric. If the player touches the enemy in this phase, the player dies. When the “electric” animation ends, the enemy moves back up to its initial position.

The Sparky class describes this enemy. Add a class with this name to the “Enemies” folder. Again, there should be no surprises in our version of the code, so let’s go over the details very quickly:

- The constructor of Sparky requires a reference to the level (as usual), but also a parameter `basePosition`, which is the position that Sparky should have when it has fallen all the way down. The “S” symbol in a level’s text file indicates this base position.
- The `Reset` method (which is also called in the constructor) plays the “idle” animation and then sets the enemy’s position and velocity so that it floats 120 pixels above `basePosition`. Finally, it calculates a random number between 3 and 5, and it sets that as the number of seconds until the enemy will start falling down.
- The `Update` method has two parts: one for when the enemy is still floating and one for when it is falling or rising. In the first case, it decreases the timer, and it starts falling down when enough has passed. In the second case, it checks for a few different things: the enemy should stop falling when it has reached `basePosition`, it should move back up when the “electrocute” animation has ended, and it should call `Reset` when it has moved all the way back up. Finally, if the enemy is currently deadly and the player is touching it, the player should die.
- The `IsDeadly` property returns whether or not the enemy can currently kill the player. This is true when the enemy is playing the “electrocute” animation *and* if that animation is between the 11th and 27th frame. If you look at the sprite `spr_electrocute@6x5`, you’ll see that those are the frames in which Sparky is surrounded by electricity.

You don’t have to change any other classes to let this code work. To load this enemy in the correct way, add the following method to *LevelLoading.cs*:

```
void LoadSparkyEnemy(int x, int y)
{
    Sparky s = new Sparky(this, GetCellBottomCenter(x,y));
    AddChild(s);
}
```

and call it when the program reads the symbol “S”. Just like `LoadTurtleEnemy`, this method will give the enemy a base position that lies at the bottom center of its tile.

If you want to see this enemy in the game, compile and run the program again and have a look at levels 8–10.

25.3 Patrolling Enemies

The enemies we’ve added up until now are pretty simple: they always do the same thing, and their behavior does not depend on where they are in the level or on what the player is doing. Let’s add some enemies that are slightly smarter: *patrolling* enemies that can move back and forth on a platform, possibly depending on the player’s current position. Using inheritance, we’ll set up a few variants of these enemies that all behave a little bit differently.

25.3.1 The Basic *PatrollingEnemy* Class

The basic version of this enemy is described in the class `PatrollingEnemy`. This enemy walks back and forth on a platform, and it waits at the edge of a platform for half a second before it moves around. To make it easier for you to develop subclasses later on, it may be a good idea to simply copy our version of `PatrollingEnemy`. Here's an overview:

- In the constructor of `PatrollingEnemy`, we store references to the level and to the starting position, and we load the main animation for this enemy character (which is an angry-looking flame). We then call the `Reset` method to apply some more initial settings.
- The `Reset` method starts playing the animation, sets the object's origin, moves the object to its starting position, and lets the object start moving to the right.
- The `Update` method contains a few cases again. If the enemy is currently waiting at the edge of a platform, it decreases the `waitTime` timer and checks if it can move around. If the enemy is *not* waiting, it checks if it has reached the edge of a platform; if that happens, then the enemy should start waiting here. Finally, a pixel-precise collision with the bomb character should cause the bomb character to die, no matter what state the enemy is in.
- The `TurnAround` method gives the enemy a horizontal speed in the other direction, and it mirrors the enemy's sprite.
- The most interesting method here is `CanMoveForward`. This method checks if a position in front of the enemy is not blocked by a wall *and* has a platform/wall tile below it. These checks use the same `GetTileType` method of the `Level` class that we also used for `Player` in the previous chapter. Depending on the enemy's walking direction, we check the *bottom-left* or *bottom-right* position of the enemy's bounding box. Take your time to understand how this `CanMoveForward` method works.

Note that some methods and member variables in this code are marked as **protected**. That's because we will create *subclasses* of `PatrollingEnemy` that will need access to some of this data and behavior.

25.3.2 Subclasses for Variants of the Enemy

Next, let's create some variants of this enemy, in the form of subclasses. The only method that these subclasses will override is the `Update` method. Everything else (apart from a custom constructor, of course) can be reused from the `PatrollingEnemy` class. In the following text, we'll only discuss the differences between these classes and their superclass. Try writing the rest of the classes yourself, keeping in mind that they should be subclasses of `PatrollingEnemy`. You can look for the answers in the `TickTick4` project.

The first subclass that we'll add is named `UnpredictableEnemy`. This enemy has all the standard patrolling behavior, with one extra feature: it can turn around at random moments. Also, whenever it turns around, it selects a new random speed between the following two values:

```
const float minSpeed = 80, maxSpeed = 140;
```

In the `Update` method, we start with basic patrolling:

```
base.Update(gameTime);
```

Whenever the enemy is not waiting, there's (in each frame) a small chance that the enemy turns around:

```
if (waitTime <= 0 && ExtendedGame.Random.NextDouble() <= 0.01)
{
    TurnAround();
    ...
}
```

You've seen a similar trick in the Painter game, where we let the paint cans reappear at the top of the screen at random moments. After turning around, we calculate a new speed between minSpeed and maxSpeed:

```
float randomSpeed = minSpeed
    + (float)ExtendedGame.Random.NextDouble() * (maxSpeed - minSpeed);
```

and we let the enemy move at that speed, keeping the direction it already had:

```
velocity.X = Math.Sign(velocity.X) * randomSpeed;
```

The Math.Sign method is probably new to you. This method returns `-1` if its input value is negative, `1` if the input value is positive, and `0` if the input value is zero. In this case, we use it to make sure the flame's direction stays the same. Feel free to take some time to understand this part of the code.

The second subclass is named `PlayerFollowingEnemy`. Compared to a regular patrolling enemy, this enemy has the feature that it will move towards the player whenever the player moves. In the `Update` method, we start with regular patrolling again:

```
base.Update(gameTime);
```

Next, we check if the player is moving, and if the enemy is not already waiting itself:

```
if (level.Player.IsMoving && velocity.X != 0)
{
    ...
}
```

The `IsMoving` property of `Player` does not exist yet. Go ahead and add it: this should be a read-only property that returns **true** when the bomb character's velocity is zero and **false** otherwise. (By the way, you could also choose to put this property in the `GameObject` class, all the way up in the class hierarchy. Another option would be to make the velocity of a `GameObject` publicly readable.)

Why do we add this extra condition that the player should be moving? Well, if you leave this condition out, the enemy becomes a bit too intelligent, and you could design levels in which some water drops are impossible to collect. In this case, we'll intentionally make the enemy "dumber" to make the gameplay more interesting. When playing the game, players will (hopefully) notice that the behavior of this enemy depends on their own movement. Discovering how the enemy reacts to their actions becomes part of the game!¹

Anyway, at the position of "...", we should check if the player is on a different side than the direction in which the enemy is now moving. If that's true, then the enemy should turn around. The following code gets the job done:

```
float dx = level.Player.GlobalPosition.X - GlobalPosition.X;
if (Math.Sign(dx) != Math.Sign(velocity.X))
    TurnAround();
```

¹Unless the player reads this book first. Spoiler alert!

To improve this code even further, add the following to the **if** condition:

`&& Math.Abs(dx) > 100`

This will make sure that the enemy always moves a bit before turning around. Otherwise, the enemy might flip back and forth very quickly when the player is above or below this enemy.

25.3.3 Adding the Enemies to the Game World

The three variants of the flame enemy are now finished. To load these enemies from a file, add another method `LoadFlameEnemy` to the file *LevelLoading.cs*. Give this method a *third* parameter that stores the symbol we want to load (which will be “A”, “B”, or “C”). Depending on that symbol, we’ll create one of the three enemy types:

```
void LoadFlameEnemy(int x, int y, char symbol)
{
    Vector2 pos = GetCellBottomCenter(x, y);

    PatrollingEnemy enemy;
    if (symbol == 'A')
        enemy = new PatrollingEnemy(this, pos);
    else if (symbol == 'B')
        enemy = new PlayerFollowingEnemy(this, pos);
    else
        enemy = new UnpredictableEnemy(this, pos);

    AddChild(enemy);
}
```

Remember: thanks to polymorphism, it’s possible to store all three types of enemies in a variable of type `PatrollingEnemy`. Finally, change the `AddTile` method so that it calls the `LoadFlameEnemy` method when you encounter the symbol “A”, “B”, or “C”.

That’s it: all enemies of Tick Tick have now been added to the game!



CHECKPOINT: TickTick4

You should now be able to play all levels and see all enemy types in action. (Remember that the flame enemies come in three variants. As a player, you don’t immediately see which variant you’re dealing with.)

Enemy Software Architecture — As you can see, all these different types of enemies are slightly different, but they have a lot of things in common as well. We’ve used inheritance to encode the similarities between the three flame enemies, but even those still have quite some code in common with the *other* enemies. In other words, you can probably design an even better way to organize these enemies into classes. For example, you could think of a general `Enemy` class that already contains a reference to the level. This general class might also contain a general system for switching between different states and for checking if the enemy collides with the player.

(continued)

In artificial intelligence, a general structure for switching between states is called a *finite state machine*. If you're up to the challenge, try to write reusable code for creating a finite state machine, and then try rewriting the existing enemy classes to use it.

We've now defined a few different kinds of enemies, with varying intelligence and capabilities. Feel free to create other enemies that are smarter (or dumber?), less predictable, or more interesting in other ways.

In this chapter, we did not apply any *physics* to the enemies. If you want to start building smarter enemies that can fall and jump, you'll need physics just like we implemented for the player. In that case, it's good to move some code from the `Player` class to a more reusable place in the program. How about adding an abstract `Character` class that becomes the superclass of `Player` and all enemies?

But even without physics, you could add many interesting features to your enemies. Can you let an enemy move faster when the player is nearby? Or can you let an enemy shoot bullets towards the player? The possibilities are endless, so go ahead and try things out for yourself!

25.4 What You Have Learned

In this chapter, you have learned:

- how to define different kinds of enemies that switch between various states;
- how to use inheritance to create variety in the behavior of enemies.

25.5 Exercises

1. *Improving the Class Hierarchy*

There are quite some similarities between the different enemies of Tick Tick. It's time to improve the code and to train your skills with inheritance again.

- a. Add an abstract class `Enemy` that will become the superclass of all specific enemies. Give this class the code that all enemies currently have in common. Note: as usual, you can fill in the details in many different ways, and there's not a single "best" answer.
- b. If you think about it, there are actually also similarities between enemies and the `Player` class. What are these similarities, and how could you encode this "overlap" in the program?

Chapter 26

Finishing the Game



In this final chapter of the book, you'll finish the game Tick Tick. First, you'll update the game so that the player can collect water drops and finish a level. Next, you'll deal with resetting the level when the player dies. After that, we'll add a timer to a level. We'll finish off by adding visual effects, sound, and music.

Because this is the final chapter, most of the work for this chapter should be easily doable for you by now. So, we won't give many code examples here; instead, we invite you to write most of the code yourself. You can always open the TickTickFinal example project to see our version of the final program.

At the end of this chapter, you'll have created the complete final version of Tick Tick! We'll then take a moment to look back at the book, and we'll discuss some possible next steps for you as a (game) programmer.

26.1 Level Progression

The first feature that we're going to add is *level progression*: players should be able to collect all water drops, finish the level, and move on to the next level.

26.1.1 Collecting Water Drops

Let's make it possible for the player to collect the water drops in a level. If the bomb character collides with a water drop, then that drop should become invisible. The WaterDrop class is a logical place for this new code. In the Update method of WaterDrop, add the following:

```
if (Visible && HasPixelPreciseCollision(level.Player))  
    Visible = false;
```

This only works if you give WaterDrop a reference to the current level, just like we've done for the Player class and all enemies. You should be able to do that on your own by now.

26.1.2 Checking If the Level Has Been Completed

The player should be able to complete a level by collecting all water drops and then reaching the goal object. Therefore, somewhere in the program, you need to check if the bomb character collides with the goal object *and* all water drops have been collected. According to us, the `Level` class is the nicest place to perform this check, because this class knows a lot about all objects in the level.

First, give the `Level` class a property `AllDropsCollected` that returns whether or not all `WaterDrop` objects in the level are currently invisible. You can find this out with a simple **foreach** loop over the `waterDrops` list. Next, give `Level` a custom `Update` method that (besides calling `base.Update`) contains the following code:

```
if (!completionDetected && AllDropsCollected && Player.HasPixelPreciseCollision(goal))
{
    completionDetected = true;
    ExtendedGameWithLevels.GetPlayingState().LevelCompleted(levelIndex);
    ...
}
```

This will mark the level as “completed” as soon as the player truly finishes this level. Make sure to add a new member variable `completionDetected`, which keeps track of whether we’ve reached this part of the code already. By including that member variable here, the `LevelCompleted` method will get called only once and not many frames in a row.

26.1.3 Celebrating a Victory

The example assets contain a “celebration” animation for the bomb character. Let’s show this animation when the player finishes a level. To do that, give the `Player` class a member variable `bool isCelebrating` that is initially **false**. Also give it the following method:

```
public void Celebrate()
{
    isCelebrating = true;
    PlayAnimation("celebrate");
    SetOriginToBottomCenter();

    // stop moving
    velocity = Vector2.Zero;
}
```

Make sure to call this method when the player completes the level (i.e., at the position of “...” in the previous code fragment).

When the `Player` object is celebrating its victory, it should no longer respond to keyboard input. Change the `HandleInput` method so that it immediately returns when `isCelebrating` has the value **true**. Furthermore, when we’re celebrating, the `Update` method of `Player` should skip all code related to horizontal movement. Change the `Update` method so that it looks something like this:

```
public override void Update(GameTime gameTime)
{
    Vector2 previousPosition = position;

    if (!isCelebrating)
    {
        // all existing code related to friction and velocity.X
    }
}
```

```

    }
    else
        velocity.X = 0;

    ApplyGravity(gameTime);
    base.Update(gameTime);

    HandleTileCollisions(previousPosition);
}

```

That way, when we've finished a level, the character will still fall down onto the nearest ground tile, but it will not move horizontally anymore.

If you compile and run the game now, you'll be able to finish a level. After that, pressing the spacebar should automatically lead you to the next level (if it exists). That part was already prepared for you in the TickTick1 project from Chap. 22.

26.2 Life and Death

For our next feature, let's make sure that the bomb character can die, after which the player should be allowed to reset the level.

26.2.1 Changing the *Player* Class

Start by giving the Player class the following property:

```
public bool IsAlive { get; private set; }
```

and set this property to **true** in the constructor of Player. The IsAlive property will allow other objects to ask whether the Player is alive, but only the Player class itself will be allowed to *change* that value. (By the way, you can achieve the same effect with a private member variable and a public read-only property, as we've done many times throughout this book.)

The Player class already contains an empty Die method. It's time to fill it in now, so that it changes the IsAlive property and then plays an animation:

```
public void Die()
{
    IsAlive = false;
    PlayAnimation("die");
    velocity = new Vector2(0, -jumpSpeed);
}

```

The last instruction of this method will let the character start moving up in a straight line. This is because when the character dies, we want it to bounce up and then fall down through the level. To achieve that effect, you should do a few more things:

- At the end of the Update method, make sure that HandleTileCollisions only gets called when the character is alive.
- Give the Player class a read-only property **bool** CanCollideWithObjects, which only returns **true** when the character is still alive *and* it's not celebrating.
- Change the HandleInput method so that it does nothing when CanCollideWithObjects is **false**.

- Change the Update method so that the friction-related code only gets executed when CanCollideWithObjects is **true**.

Finally, we can now also make sure that the character dies when it falls down past the bottom of the level. At the end of Update, add the following code:

```
if (BoundingBox.Center.Y > level.BoundingBox.Bottom)
    Die();
```

This will make sure that players can still see the death animation just before the character leaves the screen. (Of course, only execute this code when the character is still alive.)

26.2.2 Changing Other Classes

Now that a Player object knows how to die, we can make more changes elsewhere:

- In all enemy classes that currently interact with the player in some way, change the code so that this interaction only happens if CanCollideWithObjects is **true**. That way, the character can (for example) no longer die after the player has finished the level, and the character won't keep bouncing on the same enemy after it has died.
- Likewise, in the WaterDrop class, only check for collisions with the bomb character when CanCollideWithObjects is **true**.
- In the PlayingState class, the “game over” overlay should be shown whenever the bomb character is dead. There are many ways to do that, but the easiest way is to add a single line to the Update method there:

```
gameOverOverlay.Visible = !level.Player.IsAlive;
```

- In the HandleInput method of PlayingState, add a third case for when the player is not alive. In that case, pressing the spacebar should reset the level.

If you compile and run the game now, you'll see that the bomb character can die at the right moments. However, resetting the level doesn't work very well yet.

26.2.3 Resetting the Level

The Level class already inherits a Reset method from its superclass (GameObjectList). So, by default, resetting the level will call the Reset method of all objects *within* the level. This behavior is exactly what we want, but we still need to give some classes their own Reset method that resets the proper variables.

There are only three classes that still need a custom Reset method. The first class is Level itself. This method should call **base.Reset** (to let all other objects reset themselves) and set the completionDetected variable back to **false**.

The second class is WaterDrop: this class should set the Visible property back to **true**, and it should return the object to its starting position. To enable that, it's a good idea to give the WaterDrop constructor a second parameter (Vector2 startPosition), so that you can remember the starting position and easily go back to it.

The third class is `Player`. Its `Reset` method should return the player to its starting position and stop the character's movement. Other than that, it should do many of the things that are currently still inside the *constructor* of the `Player` class. In total, the `Reset` method should look like this:

```
public override void Reset()
{
    // go back to the starting position
    localPosition = startPosition;
    velocity = Vector2.Zero;
    desiredHorizontalSpeed = 0;

    // start with the idle sprite
    PlayAnimation("idle", true);
    SetOriginToBottomCenter();
    facingLeft = false;
    isGrounded = true;
    standingOnIceTile = standingOnHotTile = false;
    IsAlive = true;
    isCelebrating = false;
}
```

In the constructor of `Player`, you can now replace many instructions by a single call to the `Reset` method. Please do this now. Also, `startPosition` is a new member variable that you should add, just like in the `WaterDrop` class position.

All other classes (such as the enemies) already have the correct resetting behavior. If you compile and run the game again, you should now be able to fully reset a level when the player dies. Take your time to check if all objects really go back to their initial state. In the end, resetting a level should have the same effect as going back to the menu and loading the same level again.

26.3 Adding a Timer

We'd almost forget it, but the main idea of *Tick Tick* game is that the bomb character should explode after a certain amount of time. So, the game should keep track of how much time the player has left, and this remaining time should be shown somewhere on the screen. You'll implement that in this section.

26.3.1 The *BombTimer* Class: Maintaining and Showing the Time

In the "LevelObjects" folder, add a class `BombTimer` that inherits from `GameObjectList`. The class should have a member variable **double** `timeLeft` (which will keep track of the remaining seconds for the level) and a property **bool** `Running` that can be got or set publicly. (Soon, we'll make sure that the timer only goes down whenever `Running` is true.) Also add a public read-only property **bool** `HasPassed` that returns whether or not `timeLeft` is zero or lower. Finally, add a member variable `TextGameObject` `label`, which will store an extra reference to the text that's shown on the screen.

The constructor of `BombTimer` should add two child objects (a background image and the label), set its position in the level, and call the `Reset` method:

```
public BombTimer()
{
    localPosition = new Vector2(20, 20);

    SpriteGameObject background = new SpriteGameObject(
        "Sprites/UI/spr_timer", TickTick.Depth_UIBackground);
    AddChild(background);

    label = new TextGameObject("Fonts/MainFont", TickTick.Depth_UIForeground,
        Color.Yellow, TextGameObject.Alignment.Center);
    label.LocalPosition = new Vector2(50,25);
    AddChild(label);

    Reset();
}
```

The `Reset` method should set `Running` to **true** and set `timeLeft` to 30 s.

As usual, the `Update` method is the most interesting one for this class. If `Running` is **false**, then this method does nothing. Otherwise, if the time has not yet run out, this method should decrease `timeLeft` by the amount of time that has passed.

Next up, the method should update the label object so that the time is correctly drawn on the screen. We want to draw a text with the format “mm:ss”, where “mm” indicates the number of minutes and “ss” indicates the number of seconds. Start with the following instruction:

```
int secondsLeft = (int)Math.Ceiling(timeLeft);
```

This code uses the *ceiling* function to round the remaining time up to the nearest integer. That way, we’ll always include the currently passing second in the number that we’ll draw. As a result, the text “00:00” will only be shown when the timer has really passed. To set the label, use the following instruction:

```
label.Text = CreateTimeString(secondsLeft);
```

where `CreateTimeString` is a helper method that we still need to add. Also, to impose some extra stress onto the player, let’s change the *color* of the text so that it blinks between red and yellow in the last 10 s. The following code does this, making clever use of the modulo operator:

```
if (secondsLeft <= 10 && secondsLeft % 2 == 0)
    label.Color = Color.Red;
else
    label.Color = Color.Yellow;
```

What should the `CreateTimeString` method look like? Well, the expression `secondsLeft / 60` gives us the number of full *minutes*, and the expression `secondsLeft % 60` gives us the remainder in seconds. But we cannot draw these numbers immediately: for instance, if there are only 5 s left, we want to show the text “00:05” and not the text “0:5”. Luckily, the `String` class has a nice method `PadLeft(int length, char ch)`, which keeps adding the character `ch` in front of a string until that string becomes `length` characters long. In this case, we need to call `PadLeft(2,'0')` to possibly add a zero in front of a number. This leads to the following `CreateTimeString` method:

```
string CreateTimeString(int secondsLeft)
{
    return (secondsLeft / 60).ToString().PadLeft(2, '0')
        + ":" + (secondsLeft % 60).ToString().PadLeft(2, '0');
}
```

26.3.2 Integrating the Timer into the Game

To add our new timer to the game, give the `Level` class a member variable `BombTimer timer`. In the constructor of `Level`, initialize this object and add it to the game world.

The `Update` method of `Level` should use the timer in two ways. First of all, in the current `if` block that marks the level as “completed,” you should set `timer.Running` to **false**, so that the timer will freeze as soon as the player finishes the level. Second, after that `if` block, we should check if the timer has just passed. In that case, the player should die. The following code does this:

```
else if (Player.IsAlive && timer.HasPassed)
    Player.Explode();
```

Here, `Explode` is a yet-to-be-made variant of the `Die` method, which should show an explosion animation instead of the regular death animation. When the bomb character is exploding, it should behave slightly differently than usual: it should stay at the same position without falling down due to gravity. To make this possible, give the `Player` class a member variable **bool** `isExploding`, which gets set to **false** by the `Reset` method. Then change the `Update` method so that gravity only gets applied when the character is not exploding. Finally, add the `Explode` method, which sets some variables and starts an animation:

```
public void Explode()
{
    IsAlive = false;
    isExploding = true;
    PlayAnimation("explode");
    velocity = Vector2.Zero;
}
```

If you compile and run the game now, all levels should feature a 30-s timer that counts down. When this time has passed, the bomb character will explode, and the regular “game over” screen will be shown (thanks to the `IsAlive` property).

For completeness, it’s a good idea to stop the timer when the player dies for any reason that is *not* time-related. Add an instruction for that in the `Die` method now.

26.3.3 Letting Hot Tiles Speed Up the Timer

Do you remember the hot tiles that can appear in a level? Whenever the bomb character is standing on such a tile, the level’s timer should go faster. We’ve already prepared the `Player` class with a member variable `standingOnHotTile` that gets updated in each frame, but we weren’t *using* that variable for anything yet.

To make the timer go faster or slower, you need to do the following things:

- In the `Level` class, convert the timer to a property `Timer`, with the same accessibility details as `Player`.
- Give the `BombTimer` class a public property **float** `Multiplier` that can be got and set. In the `Reset` method, set its value to 1. In the `Update` method, take this multiplier into account when decreasing the timer:

```
timeLeft -= gameTime.ElapsedGameTime.TotalSeconds * Multiplier;
```

- In the `Player` class, you can now write `level.Timer.Multiplier` to access the multiplier. Near the end of the `Update` method, just after the call to `HandleTileCollisions`, set the multiplier to 2 when the character is standing on a hot tile, and set it to 1 otherwise.

26.4 Finishing Touches

We're now getting very close to finishing the Tick Tick game.¹ Functionally, the game's already finished! In this section, you'll add some final features: a nicer-looking background and sound effects.

26.4.1 Adding Mountains in the Background

To make the level background a bit more interesting, let's add mountains and clouds to it. We'll do this at the end of the `Level` constructor.

First, let's add a few mountains at the bottom of the level, with a certain horizontal distance between them. There are two different mountain sprites (`spr_mountain_1` and `spr_mountain_2` in the 'Sprites/Background' folder); let's choose randomly between these two for every mountain that we add. Also, we want to draw each mountain at a slightly different depth, but all mountains should be far enough in the background.

The following code adds four mountains to the `backgrounds` object that the `Level` constructor was already making:

```
for (int i = 0; i < 4; i++)
{
    SpriteGameObject mountain = new SpriteGameObject(
        "Sprites/Backgrounds/spr_mountain_" + (ExtendedGame.Random.Next(2) + 1),
        TickTick.Depth_Background + 0.01f * (float)ExtendedGame.Random.NextDouble());

    ...
    backgrounds.AddChild(mountain);
}
```

Indirectly, this will add the mountains to the game world. At the position of "...", we still need to give each mountain a position. The following code moves each mountain a certain amount to the right, while still letting the mountains overlap:

```
mountain.LocalPosition = new Vector2(
    mountain.Width * (i - 1) * 0.4f,
    BoundingBox.Height - mountain.Height);
```

The `y` component of `BoundingBox.Height - mountain.Height` ensures that each mountain aligns nicely with the bottom of the level. Remember: the `BoundingBox` property of `Level` only works *after* we've loaded a level file, because otherwise the level doesn't know its own width and height yet. That's why we have to add this code at the *end* of the `Level` constructor.

Run the game again, and you'll see different mountains each time you start a level.

26.4.2 Adding Moving Clouds

For clouds, we have something more advanced in mind: we want the clouds to have random sprites from a selection of five cloud sprites, they should move horizontally through the screen at random heights, and they should re-spawn with a new random sprite and height after they've moved out of the

¹Can you feel the timer approaching 00:00?

screen completely. Because this is quite a lot of special behavior, it makes sense to add a new `Cloud` class to describe an individual cloud. We'll give you a short overview of what this class should do; try to work out the details yourself, or look at the `TickTickFinal` example project.

The `Cloud` class should inherit from `SpriteGameObject`. Give this class a member variable that refers to the level, and (as usual) give it a constructor that receives this `Level` object as a parameter. When calling the base constructor, you can use `null` as a default sprite and `TickTick.Depth_Background` as a default depth.

Add a method `void Randomize()` that will give the cloud a new random sprite, speed, and direction. This method should first calculate a random depth (`TickTick.Depth_Background` plus a random number between 0 and 0.2). Next, it should set the sprite to a new random cloud image:

```
sprite = new SpriteSheet(
    "Sprites/Backgrounds/spr_cloud_" + ExtendedGame.Random.Next(1, 6),
    depth);
```

where `depth` is the random depth you've just calculated. After that, calculate a random *y*-coordinate, so that the *bottom* of the sprite will have a *y*-position between 100 and 600. Also calculate a random speed between 10 and 50. Finally, decide randomly between two cases. In the first case, the cloud should receive a position just outside the screen on the left side (combined with the random *y*-coordinate you've just calculated), and it should start moving to the right (with the random speed you've just calculated). In the second case, the cloud should start outside the screen on the *right* side (based on the `BoundingBox` of the level) and then move to the *left*.

Once you've completed the `Randomize` method, you can fill in the rest of the class. Override the `Update` method so that it first calls `base.Update` (which will make the cloud move) and then checks if the cloud has moved out of the screen. If that has indeed happened, call `Randomize` again to let the cloud start over with new random settings. Note that there are two different ways in which the cloud can leave the screen, depending on its movement direction.

Finally, override the `Reset` method so that it first calls `Randomize` and then gives the cloud a random *x*-coordinate inside the level's bounding box. Call this `Reset` method in the constructor. This will make sure that the clouds are already visible when the level starts/resets, as if they've already been moving around for a while.

When the `Cloud` class is complete, you can add a few clouds to a level at the end of the `Level` constructor:

```
for (int i = 0; i < 6; i++)
    backgrounds.AddChild(new Cloud(this));
```

Compile and run the game again to see the clouds in action. Feel free to make changes to their behavior and appearance if you want to.

26.4.3 Adding Sound Effects

The very last step is to play sound effects at certain moments, using this instruction:

```
ExtendedGame.AssetManager.PlaySoundEffect(...);
```

The example assets contain several sound effects. The purpose of most of them is quite straightforward:

- *snd_won* should play when the player finishes a level. (This is already handled by the starting code we've provided for you, in the *PlayingState* class.)
- *snd_watercollected* should play whenever a water drop gets collected.
- *snd_player_jump* should play when the bomb character jumps.
- *snd_player_explode* should play when the bomb character explodes.
- *snd_player_die* should play in other cases where the bomb character dies (i.e., when it falls or when it touches an enemy).

The final two sound effects are a low and a high beep sound: *snd_beep* and *snd_beep_high*. These sound effects are meant to indicate that the level's timer is about to expire. When the timer is at 10 s or less, the low beep sound should be played once per second. In the last 3 s, the high beep sound should be played instead.

To make this possible, the *BombTimer* class should recognize when the displayed number of seconds has just changed. In the *Update* method, just before decreasing the *timeLeft* variable, store the old time as a backup:

```
double oldTimeLeft = timeLeft;
```

At the end of *Update*, the following *if* condition checks if the displayed number of seconds has changed in this frame:

```
if ((int)Math.Ceiling(oldTimeLeft) != secondsLeft)
{
    ...
}
```

And inside that *if* block, play either of the sounds when there are at most 10 s left:

```
if (secondsLeft <= 3) // high beep
    ExtendedGame.AssetManager.PlaySoundEffect("Sounds/snd_beep_high");
else if (secondsLeft <= 10) // low beep
    ExtendedGame.AssetManager.PlaySoundEffect("Sounds/snd_beep");
```

Of course, feel free to add other sound effects. What about a sound when a rocket enemy spawns or when the Sparky and turtle enemies attack?



CHECKPOINT: TickTickFinal

Congratulations: you've reached the very last checkpoint of this book. The Tick Tick game is now finished!

26.5 What You Have Learned

In this chapter, you have learned:

- how to properly reset a level when the player's character dies;
- how to add a timer to a level;
- how to create animated backgrounds.

In this part of the book, we’ve shown you how to build a platform game with commonly occurring elements such as collecting items, avoiding enemies, game physics, going from one level to another, and so on. Along the way, you’ve learned about *exceptions*, and you’ve seen more examples of communication between classes and objects in a large project.

There are many things that you could still add to the Tick Tick game: more levels, more enemies, more different items to pick up, more sound effects, and so on. With the techniques you’ve learned in this book, those extensions should not be too difficult to create. You could also decide to work on things that go a bit further: playing with other players over a network, adding a side-scrolling camera, and playing in-game movies between levels, and you can probably think of other interesting extensions.

The exercises for this chapter give a number of suggestions in increasing order of difficulty. But in the end, your only limit is your own creativity!

26.6 Exercises

1. *Adding Side Scrolling to Tick Tick*

The goal of this exercise is to extend the Tick Tick game so that levels can be larger than a single screen. You’ll work towards a version of Tick Tick in which levels can have any size. There will be a camera object that shows a certain part of the level, and this camera should follow the bomb character.

- a. The first step is to extend the game engine so that you can define a *camera*. This camera defines which part of the world is currently being shown on the screen. Add a Camera class to the Engine library that decides which part of the world is currently being shown, for example, by using a Rectangle. When drawing objects on the screen, you’ll now also have to take the position of the camera into account.

To test your camera, try to run the program with a few different values for the camera position. Make sure that the camera works correctly in full-screen mode as well.

- b. The next step is to allow levels to have any width and height, just like the puzzles from Penguin Pairs. Extend the game so that this works without problems. You may assume that a level is never smaller than the levels we already have.
- c. Finally, to really add side-scrolling behavior, the camera should *follow* the bomb character. There are a few different possibilities here. One option is to always (try to) show the character in the center of the screen. Another option is to keep the camera steady until the character crosses a certain boundary, such as at one-third and two-thirds of the visible screen. Extend the game so that the camera does one of these two things. Make sure that the camera never reaches outside the game world: for example, if the character runs into the far edges of the level or falls into a hole below the level, the camera shouldn’t move too far along with it.
- d. Make sure that some objects are *not* sensitive to the camera’s position. For example, the UI elements drawn on the foreground should always stay in place, no matter where the camera is located.

You may also want to create two different versions of the method that translates screen coordinates to world coordinates: one that uses the camera’s position and one that ignores it. Make sure that the player can still correctly click on objects that are clickable.

- e. Since our levels can now be much bigger, 30 s may not be enough time for the player to reach the exit. Extend the game so that each level (file) specifies the number of seconds that the player has when the level starts. Create a few different levels that use this ability.
- f. To create the illusion that the game is three-dimensional, it’s fun to let objects in the background “scroll more slowly” than objects in the foreground. This effect is called *parallax scrolling*: look

up some videos on the Internet to see what we mean. Implement a parallax scrolling effect for the mountains in the background. The depth of a mountain should determine how slowly the mountain moves along with the camera.

2. *Other Extensions for Tick Tick*

Below are more suggestions for things you could add to the Tick Tick game. Once again, we've ranked them by difficulty.

- a. (*) Extend the Rocket class so that when the bomb character jumps onto the rocket, the rocket dies.
- b. (**) Add an object to the game that makes the bomb character walk much faster or slower for a while. This object could be a tile or an item that can be picked up.
- c. (**) Add a health indicator for the player. Every time the bomb character touches an enemy, or if the character falls down from more than three tiles high, their health gets reduced by a certain amount. If the health reaches zero, the player dies. Add health packs to the game, so that the player can restore part of their health again.
- d. (**) Add a "shield" item to the game that (temporarily) protects the player from taking damage.
- e. (***) Add shooting behavior to the player. For example, you could allow the bomb character to shoot a bullet straight forward. If this bullet touches an enemy, the bullet disappears and the enemy dies. *Hint:* It's easiest to allow only one bullet at a time and to make this bullet visible and invisible at the right moments.
- f. (****) Try to make some of the enemies smarter. For example, can you add behavior to the flames so that they can jump from one platform to another? Or can you make the rockets smarter so that they sometimes follow the player?
- g. (****) Add hidden levels to the game. By going to a particular position in a level, the player can enter a hidden level. Of course, hidden levels should also be read from a file, just like normal levels. Programming-wise, think of a good class design where these hidden levels fit in. Gameplay-wise, think of a fun way to add secret entrances to the game. Will a secret entrance be completely invisible, or will you sometimes show a hint about its location?
- h. (*****) Add moving platforms to the game. If a character (the bomb or an enemy) is standing on a moving platform, he should move along with that platform. A moving platform should move back and forth between two points in the level. Extend your level files so that you can specify these two endpoints. This extension is quite difficult because you have to make some tough design choices. Is a moving platform a tile or something else? And what consequence does this have for collision detection?
- i. (*) Games often contain extras that don't really add anything to the gameplay but that just make the game more fun. An example of such an "Easter egg" is that a character says something ridiculous when you click on it. Extend the game such that when you click on the bomb character, the character says something funny, randomly selected from a set of sound files. You can use existing sound fragments, but you could also go crazy and record your own sounds with a microphone.

3. ** Deleting Objects from the Game World*

In the game engine you've developed throughout this book, we can't really *delete* objects from the game world yet. So far, whenever we wanted an object to disappear, we simply made it invisible. This is good enough for some games, but not for all of them.

In this final exercise of the book, you'll experiment with deleting objects during the game. You'll notice that this is pretty difficult to get right!

- a. Give an example of a situation in which the "invisibility trick" is not good enough. In other words: think of a game where it's crucial that objects can truly be added and removed from the game world at any time.

- b. Give the `GameObjectList` class a method with the following header:

```
public void RemoveChild(GameObject o)
```

This method should remove the given object from the list of children, and it should reset the “parent” reference of that object to `null`. An object with a parent should now be able to write the following:

```
(Parent as GameObjectList).RemoveChild(this);
```

to remove itself from the game-object hierarchy.

- c. In `Tick Tick`, use this new method to remove a `WaterDrop` object when the player touches it. Compile and run the game again: what happens now when you collect a water drop?
- d. The problem is that we’re trying to remove an object from a list *during a loop through that same list*. In C#, if you try to do this during a **foreach** loop, an exception will be thrown. To solve this, change the `Update` method of `GameObjectList` so that it uses a regular **for** loop instead.
- e. Now that objects can remove themselves, it’s better if the `Update` method loops through the list of children *backwards*, just like in the `HandleInput` method. Why is that?
- f. There’s one more catch: the `Level` class still stores an *extra reference* to all `WaterDrop` objects. The water drops that have removed themselves from the hierarchy will still appear in this list. Can you think of a way to fix this problem? (As usual, there are multiple possibilities.)

As you can see, as soon as you allow objects to disappear from the game world, there are many extra things that you need to pay attention to. This is exactly why we’ve always been using the “invisibility trick” instead: it was the best way to keep things simple, so that we could focus on teaching you programming concepts.

26.7 What's Next?

Here we are, at the end of the book. If you’ve followed all chapters, you should now have a good understanding of the C# programming language, and you’ve programmed several games with this knowledge.

But of course, this isn’t the end of the line. With the knowledge you have now, there are many different directions in which you can go. Here are a few suggestions.

Diving Deeper Into C#: This book covered many important features of the C# language, but we didn’t have the space to talk about *everything*. For example, we didn’t discuss *generic classes and methods* (which allow to organize your code even more nicely), and we didn’t mention *delegates* (which you can use to let objects “listen” to events such as UI interactions). There are many other neat tricks in C#; very often, these tricks allow you to write the same code in a shorter way.

Actually, C# is constantly being improved by people worldwide: every once in a while, an update appears (often combined with an update of Visual Studio) in which you can do even more with this programming language. By the time you’re reading this, we bet that there are new C# features that even *we* don’t know about yet!

Trying Out Other Languages: Now that you understand the basics of C# *and* the theoretical concepts behind it (such as object-oriented programming), you’re ready to look at many other programming languages as well. In particular, the language Java is very similar to C#, with only very subtle differences. Other object-oriented languages (such as C++) may be more different, but in the end, they’re based on the same principles.

Of course, you're free to start learning a new programming language just for fun, but it helps if you have a particular goal in mind. Some languages are particularly useful for writing certain types of programs. For instance, PHP and JavaScript are important languages for websites (and HTML5 games), Java is useful for creating Android apps, and Swift is a language for iOS app development. We recommend that you first think about what kinds of applications you want to create, and then look for a suitable language. Whatever the language will be, chances are that you'll get the hang of it very quickly, thanks to what you've learned in this book.

Moving to 3D Games and Engines: The MonoGame library is especially useful for 2D games, and it is very useful for teaching newcomers how to program. After all, as a MonoGame programmer, you're in charge of drawing all game objects yourself, and you have complete control over the classes and objects in your program. Now that you have the hang of this, it might be fun to look at "bigger" game engines such as Unity3D and Unreal.

One important difference is that these engines are very useful for 3D games. As you can probably imagine, programming the behavior of objects in a 3D world is more complex than in a 2D world. It often involves more complicated mathematics, so if you decide to go for a 3D game, it definitely helps if you're not afraid of math.

Apart from the difference between 2D and 3D, another game engine might also encourage a whole different style of programming. For example, in Unity3D, it's quite common to drag and drop objects into a game world and to program their behavior in small stand-alone *scripts*. This is a totally different approach than creating a class for each game object (which you've learned in this book). In the Unreal engine, the recommended programming style is different yet again—and you'll have to get into C++ while you're at it.

Designing and Publishing Games: Now that you know how to program games, you might have lots of ambitious ideas about new games that you want to develop (and maybe even sell). If you want to become a successful (indie) game developer, it helps if you learn more about the *other* aspects of game development.

For example, *game design* is a whole separate field of study: it focuses on what makes a game fun and how you can keep players engaged. Always keep your target audience in mind: what do your future players like, and how will you make sure that they'll enjoy your game? If you allow players to test (early versions of) your game, you'll learn valuable new things. Try to avoid "tunnel vision": a game that *you* like is not necessarily a game that your *audience* likes.

Remember that games are often made by teams of multiple people: programmers, artists, sound designers, game designers, and so on. You don't have to be good at everything all by yourself. Get to know your own strengths and weaknesses, and look for people with whom you could form a nice and complete team. To find such people, it helps to be active in social networks, start your own blog, post on forums, and so on. You could check out websites such as *indiegames.com* to learn what other indie developers are doing, and you could join events such as the Global Game Jam (*globalgamejam.org*) to meet other developers and make yourself known.

After you've completed a game, *publishing* it successfully is an art of its own. Many games are being published each day, so how do you make sure that people notice your game in the crowd? Having a good network of people will definitely help you in this phase.

Finally, especially when you're still a "beginning" developer, try to keep your goals realistic. You don't have to immediately create a new AAA franchise (although it's definitely fun to daydream about such a thing!). Start with small games that people will notice and will remember.

Whatever your next steps will be, we hope that you've enjoyed this introduction to game programming in C#, and we hope that we've inspired you to do many excellent things.

Thank you for using this book, and best of luck in the future!

Glossary

abstract class	class of which you cannot directly create an instance; intended as a base class for other classes to inherit from, 338
abstract method	method without a body, defined in an abstract class, that must be implemented in a subclass, 339
access modifier	keyword such as public or private that defines how much a variable/method/etc. can be used by other classes, 108
animation	rapid display of a sequence of images to create an illusion of movement, 451
application	computer program that can be run, often including some kind of user interaction, 15
array	(object containing a) numbered row of values of the same type, 241
asset	resource used for developing a game, such as a sprite or a sound effect, 69
assignment	instruction that assigns a (new) value to a variable, 43
base class	see “superclass”, 190
bit	unit of memory that can store two different values, 50
bool	data type for (the result of) logical expressions; its only two possible values are true and false , 95
bounding volume	shape (such as a box or a circle) that encompasses an object, used for collision detection, 464
branch	part of the code that only gets executed when a certain condition holds, e.g., in an if or switch instruction, 92
byte	unit of 8 bits in memory that can store 256 different values, 50
cast	forcing an expression to get converted to a different (but related) type, 52
char	(type of a) value that represents an Unicode character, 213
child class	see “subclass”, 190
class	group of member variables, methods, and properties that describes a certain type of object, 26
code	text that describes (part of) a program in a certain programming language; usually written by a programmer, 18
collision	intersection between bounding volumes in the game world, 463
compiler	program that checks and translates a program from source code to executable code, 17
condition	expression that yields either true or false , used, e.g., in an if instruction, 92

constructor	special method, with the same name as its class, that is automatically called when creating a new class instance, 125
declaration	program fragment that introduces a name for a class, method, variable, etc., 42
derived class	see “subclass”, 190
double	(type of a) value that represents a floating-point number with double precision, 49
enum	data type with an explicitly enumerated set of possible values, 91
exception	abnormal event that may occur during program execution that can be handled in a special way, 443
executable	a program in its final form that can be run; translated from source code by a compiler, 21
expression	piece of code that has a value of a certain type, 45
false	the Boolean value used for a condition that is not true, 95
float	(type of a) value that represents a number with decimal digits (a “floating-point number”), 50
game	(in this book) computer application that gives its users a playful experience, 15
game engine	library meant to make game development easier; collection of classes that provides structures and tasks commonly used in games, 16
game loop	main loop in a game that repeatedly updates the game world and then draws the result, 30
game state	part of a game that corresponds to a certain “phase” of the game, such as a title screen, a menu, or a “game over” window, 302
inheritance	the fact that a subclass automatically contains the data and behavior of its base class, allowing to reuse code without copying it, 190
initialization	instruction that assigns a value to a variable for the first time, 43
instance	specific object in the program that has a class as its type, 27
instruction	piece of code that changes the values in the computer’s memory in some way, 9
int	(type of a) value that represents a whole number (an “integer”), 42
interface	group of method headers that serves as a “checklist” to be implemented by other classes or the way a program interacts with a user, 341
interpreter	program that checks a program from source code and executes it, 17
iteration	a single execution of the instructions inside a loop; or sometimes: the process of looping in general, 174
library	collection of classes that can be imported into other programs, to offer helpful functionality, 36
local variable	variable that can only be used in the block in which it is declared (as opposed to a “member variable”, 57
loop	repeated execution of instruction(s), e.g. in a while , for , or foreach instruction, 174

member variable	variable that belongs to an object and that can be used throughout an entire class; also called a “field”, 58
method	group of instructions with a name; often manipulates certain objects and/or returns an expression, 11
method call	instruction that tells the program to execute the instructions inside a method, 26
namespace	group of classes that can refer to each other without explicitly specifying where they come from, 35
null	the value of a reference when it is not (yet) referring to a particular object, 149
object	group of variables in memory having a class as its type; a specific instance of a class, 27
operator	symbol that combines expressions to calculate a new expression, such as “+” for adding numbers, 46
overriding a method	redefining a method in a subclass, to give a more specific version of that method, 198
parameter	expression that can be given to a method, to give that method more specific details to work with, 26
parent class	see “superclass”, 190
partial class	description of part of a class, allowing the full class to be divided over multiple files, 442
pixel	a single dot on a (computer) screen or a single dot inside a sprite, with a certain color, 70
polymorphism	the ability of a program to process objects differently depending on their class type, 203
private	access modifier used for a member that can be used only inside the class in which it is declared, 108
property	shorter variant of a getter and/or setter method for a class; can be used as a member variable, but may have extra code behind it, 131
protected	access modifier used for a variable/method/etc. that can be used only inside its own class and its subclasses, 197
public	access modifier used for a variable/method/etc. that can be used from within all other classes, 108
recursion	(programming) concept in which a method or property calls itself again inside its own body, 279
return value	value that a method delivers after its execution, 111
RGB color model	a way of using red, green, and blue components to describe many different colors, 55
scope	the part of the program in which a particular variable can be used, 57
sealed method	overridden method that cannot be redefined anymore in subclasses, 205
semantics	rules that describe the meaning of text in a (programming) language, 20
source code	the code of a program (see ‘code’), 18
sprite	two-dimensional image to be added in a larger scene, often used in 2D games, 69

static	keyword indicating that a method/variable/etc. is shared by all instances of a class, without relying on a specific instance of that class, 155
string	(type of a) value that represents a piece of text, 214
struct	wrapper for data and behavior, similar to a class, that is accessed by value instead of by reference, 150
subclass	class that inherits from another class; also called “child class” or “derived class”, 190
superclass	class from which another class inherits; also called “parent class” or “base class”, 190
syntax	rules that describe the “form” of a (programming) language, indicating what is grammatically correct, 20
tile	basic element of a two-dimensional level; “building block” from which 2D levels are composed, 374
true	the Boolean value used for a condition that is the truth, 95
variable	memory location with a name; it can store a value of a certain type, 41
vector	object denoting a point in two- or three-dimensional space; useful for describing a game object’s position, velocity, etc., 75
virtual method	method that can be redefined (“overridden”) in a subclass, 199
void	placeholder for the return type of a method that does not return a value, 111

Index

- Abstract
 - class, 338
 - method, 339
- abstract**, 338
- Access modifiers, 108, 120, 130, 197
 - for classes, 427
 - inside properties, 138
 - internal** (*see internal*)
 - private** (*see private*)
 - protected** (*see protected*)
 - public** (*see public*)
- Animation, 451
- Application, 15
- Array, 242
 - index, 242
 - multidimensional, 245
- Artificial intelligence, 481
- Aspect ratio, 233
- Assembler, 17
- Assignment, 43
- base**, 196, 200
- Base class, 190
- Bit, 50
- bool**, 93, 95
- Boolean, 93
- Bounding volume, 464
- Branch, 93
- break**
 - combined with **switch**, 397
 - inside a loop, 181
- Byte, 50
- C++, 12
- C#, 13
- Call stack, 283
- Cast, 52, 218
- catch**, 444
- char**, 213
 - escaping, 215
- Child class, 190
- Clamping, 288
- Class, 12, 26
 - base (*see Base class*)
 - body, 27
 - child (*see Child class*)
 - generic (*see Generic class*)
 - header, 27
 - hierarchy, 204
 - parent (*see Base class*)
 - partial, 442
 - subclass (*see Child class*)
 - super- (*see Base class*)
- Class type, 53, 145
- Collection, 255
- Collision detection, 170, 464
- Color, 54
- Comments, 37
- Compilation unit, 37
- Compiler, 18
- Conditional instruction, 93
- const**, 45
- Constant, 45, 248
- Constructor, 34, 54, 78, 117, 125
 - base, 196

- continue**, 181, 471
- Coordinates
 - conversion, 236
 - screen, 236
 - world, 236
- Counter, 176
- CPU, 9
- Data type, 41
 - class (*see* Class type)
 - enumerated (*see* Enumerated type)
 - nullable, 448
 - numeric (*see* Numeric type)
 - predefined (*see* Predefined type)
 - primitive (*see* Primitive type)
- decimal**, 51
- Declaration, 42
- Declarative programming, 15
- Development cycle, 21
- Dictionary, 257, 331
- Division remainder, 46
- Documentation, 38, 220
- double**, 49
- Draw, 30
- enum**, 92
- Enumerated type, 91
- Exception, 443
 - throwing, 445
- Executable, 21
- Expression, 45
- false**, 95
- Field, 59
- File I/O, 377
 - stream, 378
- float**, 51
- Floating-point number, 50
- for**, 176
- foreach**, 256
 - enumerator, 256
 - with a dictionary, 332
- Functional programming, 14
- Game asset, 69
- Game engine, 16
- Game loop, 30
 - fixed timestep, 30
 - frame, 30
 - variable timestep, 31
- Game-object hierarchy, 275
 - child, 276
 - global position, 277
 - local position, 277
 - parent, 276
- Game state, 302, 329
- GameTime, 42
- Game world, 29
- Garbage collection, 150
- Generic, 258
- Generic class, 258
- get**, 132
- GPU, 9
- GraphicsDevice, 34
- if**, 93
- Immutable, 253
- Imperative programming, 11
- Inheritance, 33, 190
- Initialization, 43
- Initialize, 33
- Instance, 27, 53, 117, 124
- Instruction, 10, 25
- int**, 42
- Interface, 341
- internal**, 120
- Interpreter, 18
- is**, 413
- Iteration, 174
- Java, 13
- Keyboard input, 104
- KeyValuePair, 332
- Lazy evaluation, 414
- Library, 36, 426
- List, 255
- LoadContent, 33
- Local variable, 57
- Logical programming, 14
- Logic operators, 96
- Loop, 174
 - counter, 176
 - nested, 179
- Member, 59
- Member variable, 58
- Memory, 9

Method, 11, 26, 107

body, 26

call, 26, 109

header, 26, 108

parameters, 26, 109

return type (*see* Return type)

return value (*see* Return value)

Method call, 116

MonoGame, 17

MonoGame Pipeline Tool, 70

Mouse input, 80

Music, 75

Namespace, 35

Nested loops, 179

new, 54

null, 149

Numeric type, 49

Object, 12, 27, 117

Object-oriented programming, 12, 107

Operator, 46

arguments, 46

arithmetic, 46

binary, 49

comparison, 95

infix, 49

logical, 96

on strings, 254

overloading, 84

postfix, 49

prefix, 49

priority, 47

unary, 49

override, 199

Parent class, 190

Parent pointer, 276

Particle system, 319

Pixel, 70

Polymorphism, 203, 271, 345

Predefined type, 215

Primitive type, 53, 145

private, 120

Procedural programming, 11

Processor, 9

Program, 10

Program layout, 37

Programming, 10

language, 10

paradigm, 11

Property, 56, 72, 132

get, 132

read-only, 133

set, 132

static, 155

protected, 120, 197

public, 120

Quick Reference

Abstract classes, 340

Access modifiers, 121

Arrays, 243

Base, 201

Comments, 38

Data access, 133

Expressions and Operators, 47, 116

file I/O

reading, 382

writing, 383

For, 177

Foreach, 257

If, 95

If and else, 99

Inheritance, 195

Instances and member variables, 118

Instances and method calls, 119

Interfaces, 343

Is, 413

Large Classes, 443

Methods, 115

Namespaces, 35

Recursion, 282

Static, 156

Switch, 398

This, 122

Try and catch, 445

Using, 36

Variable Assignment, 45

Variable Declaration, 42

Variable Scope, 59, 185

Variables, references, null, 149

Virtual and override, 199

While, 175

Random number generator, 166

pseudo-, 168

seed, 168

Rectangle, 170

Recursion, 279, 280

base case, 281

- infinite, 283
- recursive case, 281
- ref**, 148
- Region, 442
- return**, 112
- Return type, 108
- Return value, 111
- RGB color model, 55
- Scaling matrix, 229
- Scope, 57, 184, 185
 - inside loops, 184
 - of method parameters, 110
- Screen coordinates, 236
- sealed**, 206
- Semantics, 20
- set**, 132
- Sound effects, 75
- Sprite, 69
 - batch, 74
 - origin, 82
 - rotating, 86
 - sheet, 290, 349
 - transparency, 318
- SpriteBatch, 74
- static**, 155
- Stream, 378
- StreamReader, 379, 381
- StreamWriter, 379, 382
- string**, 214
- Strings, 214
 - advanced methods, 254
 - concatenation, 216
 - ToString, 254
- struct**, 150
- Subclass, 190
- Superclass, 190
- switch**, 396
- Syntax, 20
- Texture2D, 73
- this**, 121
- throw**, 445
- true**, 95
- try**, 444
- Type, 32
 - See also* Data type
- UnloadContent, 33
- Update, 30
- User interface, 354
- using**, 36
- Variable, 41
 - assignment, 43
 - declaration, 42
 - initialization, 43
 - local, 57
 - member, 58
 - scope (*see* Scope)
- Vector, 78
 - math, 84
- Vector2, 75, 78
- Viewport, 233
- virtual**, 199
- Visual Studio project, 3, 7
- Visual Studio solution, 7
- void**, 111
- while**, 174
- World coordinates, 236